

ENERGY AND RELIABILITY MANAGEMENT IN PARALLEL REAL-TIME SYSTEMS

by

Dakai Zhu

B.E, Xi'an Jiaotong University, P.R.China, 1996

M.E, Tsinghua University, P.R.China, 1999

M.S, University of Pittsburgh, 2001

Submitted to the Graduate Faculty of
Arts and Science in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2004

UNIVERSITY OF PITTSBURGH
DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Dakai Zhu

It was defended on

November 12, 2004

and approved by

Dr. Rami Melhem

Dr. Daniel Mossé

Dr. Bruce R. Childers

Dr. Raj Rajkumar

Dissertation Advisors: Dr. Rami Melhem,

Dr. Daniel Mossé

Copyright © by Dakai Zhu
2004

ABSTRACT

**ENERGY AND RELIABILITY MANAGEMENT IN PARALLEL
REAL-TIME SYSTEMS**

Dakai Zhu, PhD

University of Pittsburgh, 2004

Historically, slack time in real-time systems has been used as temporal redundancy by roll-back recovery schemes to increase system reliability in the presence of faults. However, with advanced technologies, slack time can also be used by energy management schemes to save energy. For reliable real-time systems where higher levels of reliability are as important as lower levels of energy consumption, centralized management of slack time is desired.

For frame-based parallel real-time applications, energy management schemes are first explored. Although the simple static power management that evenly allocates static slack over a schedule is optimal for uni-processor systems, it is not optimal for parallel systems due to different levels of parallelism in a schedule. Taking parallelism variations into consideration, a parallel static power management scheme is proposed. When dynamic slack is considered, assuming global scheduling strategies, slack shifting and sharing schemes as well as speculation schemes are proposed for more energy savings.

For simultaneous management of power and reliability, checkpointing techniques are first deployed to efficiently use slack time and the optimal numbers of checkpoints needed to minimize energy consumption or to maximize system reliability are explored. Then, an energy efficient optimistic modular redundancy scheme is addressed. Finally, a framework that encompasses energy and reliability management is proposed for obtaining optimal redundant configurations. While exploring the trade-off between energy and reliability, the effects of voltage scaling on fault rates are considered.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	1
1.0 INTRODUCTION	2
2.0 BACKGROUND AND RELATED WORK	6
2.1 REAL-TIME SYSTEMS	6
2.2 ENERGY AWARE COMPUTING	8
2.3 FAULT TOLERANCE	10
2.4 ENERGY EFFICIENT FAULT TOLERANCE	12
3.0 SYSTEM MODELS AND PROBLEM DESCRIPTION	13
3.1 APPLICATION AND SYSTEM MODELS	13
3.1.1 AND Application Model	14
3.1.2 AND/OR Application Model	15
3.1.3 System Models	18
3.2 POWER MODEL AND ITS EFFECTS ON ENERGY MANAGEMENT	19
3.2.1 Dynamic Power for CMOS Based Processors	19
3.2.2 A Simple Power Model	20
3.2.3 Effects of Power Model on Voltage Scaling	24
3.3 FAULT AND RECOVERY MODELS	25
3.3.1 Fault Models	25
3.3.2 Rollback Recovery and Checkpoints	26
3.4 PROBLEM DESCRIPTION AND RESEARCH OVERVIEW	26
4.0 PARALLEL ENERGY AWARE SCHEDULING	29
4.1 SCHEDULING IN PARALLEL REAL-TIME SYSTEMS	29

4.1.1	Earliest Ready Longest Task First Heuristic (ER-LTF)	30
4.1.2	Canonical Schedules	32
4.1.3	Importance of Execution Order	33
4.1.3.1	Independent Tasks	34
4.1.3.2	Anomaly of List Scheduling for Dependent Tasks	38
4.1.3.3	List Scheduling with Fixed Priority	39
4.2	STATIC POWER MANAGEMENT (SPM)	41
4.2.1	Greedy Static Power Management (G-SPM)	42
4.2.2	Uniform Static Power Management (U-SPM)	42
4.2.3	Static Power Management with Parallelism (SPM-P)	44
4.2.3.1	SPM-P for Dual-Processor Systems	44
4.2.3.2	SPM-P for M-Processor Systems	46
4.2.4	SPM for AND/OR Applications	47
4.3	DYNAMIC POWER MANAGEMENT (DPM)	47
4.3.1	Infeasibility of Simple Greedy Slack Reclamation (GSR)	48
4.3.2	Shared Slack Reclamation (SSR) for AND-model Applications	49
4.3.2.1	Two Examples	49
4.3.2.2	Offline Phase of SSR Scheme	51
4.3.2.3	On-line Phase of SSR Scheme	52
4.3.2.4	Analysis of SSR Algorithm	53
4.3.3	Shifted/Shared Slack Reclamation (S/SSR)	56
4.3.3.1	Offline Phase of S/SSR Scheme	56
4.3.3.2	On-line Phase of S/SSR Scheme	59
4.3.3.3	Analysis of S/SSR Algorithm	61
4.4	SPECULATIVE SCHEMES	62
4.4.1	A Static Speculation Scheme	62
4.4.2	An Adaptive Speculative Scheme	63
4.5	PRACTICAL CONSIDERATIONS IN ENERGY MANAGEMENT	63
4.5.1	Overhead of Frequency Adjustment	64
4.5.1.1	Time Overhead and Slack Reservation	64

4.5.1.2	Energy Overhead	67
4.5.2	Discrete Frequency Levels	68
4.5.3	Shared Memory Access Contention	69
4.6	EVALUATIONS OF ENERGY MANAGEMENT SCHEMES	69
4.6.1	Simulation Setup	70
4.6.2	Effects of Frequency Change Overhead	74
4.6.3	Effects of Discrete Frequency Levels	75
4.6.4	Energy Savings of SPM-P	76
4.6.5	Energy Savings of S/SSR and Speculative Schemes	77
4.6.5.1	Trace Based Simulations for AND-model Applications	77
4.6.5.2	Synthetic AND/OR-model Applications	78
4.6.6	Effects of The Minimum Energy Efficient Frequency	79
4.7	THEORETICAL BOUNDS: HOW MUCH BETTER CAN WE DO?	81
4.8	CHAPTER SUMMARY	84
5.0	ENERGY EFFICIENT FAULT TOLERANCE	87
5.1	ENERGY EFFICIENT ROLL-BACK RECOVERY	88
5.1.1	Simple Scheme of Re-execution (Retry)	89
5.1.1.1	Pessimism Level: The Number of Expected Faults	89
5.1.1.2	Performability of Retry Scheme	90
5.1.1.3	Expected Energy Consumption of Retry Scheme	91
5.1.2	Checkpointing and Its Applicability	92
5.1.3	Optimal Number of Checkpoints for Maximizing Performability	95
5.1.4	Optimal Number of Checkpoints for Energy Minimization	99
5.1.4.1	Expected Energy Consumption	99
5.1.4.2	Fault-Free Energy Consumption	101
5.1.5	Evaluations of Roll-Back Recovery with Checkpoints	102
5.1.5.1	Optimal Number of Checkpoints	103
5.1.5.2	Energy Efficient Regions	104
5.1.5.3	System Performability	105
5.2	OPTIMISTIC MODULAR REDUNDANCY	106

5.2.1	Optimal Frequency Setting for OTMR	106
5.2.1.1	Expected Energy Consumption	107
5.2.1.2	Fault Free Energy Consumption	109
5.2.2	Performability of OTMR	112
5.2.3	Comparison of OTMR and Traditional TMR	112
5.2.4	Optimistic N-Modular Redundancy (ONMR)	115
5.3	ENERGY EFFICIENT REDUNDANCY CONFIGURATION	117
5.3.1	Recovery Schemes with Parallel Slack	119
5.3.1.1	Restricted Serial Recovery	119
5.3.1.2	Parallel Recovery	120
5.3.1.3	Adaptive Parallel Recovery	121
5.3.1.4	Arbitrary Number of Tasks	123
5.3.1.5	Maximum Number of Tolerated Faults	124
5.3.2	Parallel Recovery and Modular Redundancy	125
5.3.3	Optimal Redundant Configurations	126
5.3.3.1	Minimize Energy with A Given Performability Goal	126
5.3.3.2	Maximize Performability with Fixed Energy Budget	128
5.3.4	Analysis Results	130
5.3.4.1	Optimal Configuration for Energy Minimization	131
5.3.4.2	Optimal Configuration for Performability Maximization .	134
5.4	INTERPLAY OF ENERGY MANAGEMENT AND PERFORMABILITY	135
5.4.1	Voltage Scaling and Fault Rates	135
5.4.1.1	Exponential Fault Rate Model	137
5.4.2	Trade-off between Energy and Performability	137
5.4.2.1	Some Numeric Results	139
5.5	CHAPTER SUMMARY	140
6.0	CONCLUSIONS	142
7.0	FUTURE WORK	147
	BIBLIOGRAPHY	149

LIST OF TABLES

3.1	Power consumption at different frequencies for Intel XScale processors.	23
4.1	Offline variables of an AND/OR-model application	58
4.2	Frequency/voltage settings for Transmeta 5400	73
4.3	Frequency/voltage setting for Intel XScale processors	73
4.4	Energy savings vs. U-SPM using trace data	78
5.1	The maximum number of faults that can be tolerated.	124
5.2	The optimal redundant configurations for different recovery schemes	131
5.3	The effects of pessimism levels on optimal redundant configuration	134

LIST OF FIGURES

3.1	AND-model application examples.	15
3.2	AND/OR structures	16
3.3	Loop expansion in AND/OR-model applications.	17
3.4	An example of AND/OR-model applications.	18
3.5	The illustration of a simple power model	22
3.6	Power model validation using power and frequency numbers of Intel XScale .	23
3.7	Summary of solutions and research overview.	27
4.1	Canonical schedules of an AND-Model application	32
4.2	Canonical schedule of an AND/OR-Model application	33
4.3	The canonical schedule and one running of an independent task set	34
4.4	The canonical schedule and one running of one dependent task set	38
4.5	Fixed-priority list scheduling for dependent tasks	39
4.6	A simple example and its canonical schedule	42
4.7	Two simple static energy management schemes	43
4.8	Parallelism in the schedule for a simple application	44
4.9	The simple greedy scheme	48
4.10	SSR for independent tasks	50
4.11	SSR for dependent tasks	51
4.12	The shifted canonical schedules for an AND/OR application	58
4.13	An actual execution of the AND/OR application	61
4.14	Slack reservation for incorporating frequency adjustment overhead.	65
4.15	Slack is not enough for an additional frequency change.	67

4.16 Slack sharing with frequency change overhead.	67
4.17 Dependence graph for a synthetic AND/OR-model application.	70
4.18 ATR and its execution time	71
4.19 MPGE-1 and its execution time	72
4.20 The effects of frequency change overhead on energy savings.	75
4.21 The effects of discrete frequencies on energy savings.	76
4.22 The normalized energy vs. <i>LDR</i> for different SPMs.	77
4.23 Energy savings vs. execution time variations	79
4.24 Effects of the minimum energy efficient frequency	80
4.25 The theoretical bounds on energy savings.	83
5.1 The retry scheme	90
5.2 Checkpoints and recovery sections	94
5.3 Performability and number of recovery sections.	98
5.4 Optimal number of checkpoints for Duplex; $\alpha = 0.1$	103
5.5 Energy efficient regions for Duplex and TMR	104
5.6 The probability of failure ($1 - \text{performability}$) for Duplex and TMR.	105
5.7 Optimal frequency settings for OTMR	107
5.8 Optimal frequencies for OTMR and TMR	114
5.9 The energy consumption of OTMR and TMR	114
5.10 The performability of OTMR and TMR	115
5.11 Slack and temporal redundancy in parallel systems	118
5.12 Different recovery schemes.	119
5.13 The faults tolerated by different recovery schemes	120
5.14 The minimum expected energy consumption for different recovery schemes.	132
5.15 The minimum expected energy consumption under different system load	133
5.16 The faults tolerated with limited energy budget	134
5.17 The performability and expected energy consumption for different values of d	139

ACKNOWLEDGEMENTS

This dissertation is the fruit of years of hard work as a member of the Power Aware Real-Time Systems (PARTS) research group. It is the result of much time and great effort not only on my part, but also on the part of my advisors, committee members, colleagues and my family. I would like to take this chance to express my sincere thanks to all of them!

First of all, I am grateful to my advisors, Prof. Rami Melhem and Prof. Daniel Mossé for their guidance, encouragement and support throughout my Ph.D. studies. Despite their busy schedules, they were always willing to discuss and provide help on every matter. Having selected an academic career, I cannot help thinking how much I would be happy if I were able to contribute to my (future) students as much as they did to me.

Special thanks to Dr. Bruce R. Childers for his help and valuable suggestions, especially during the early stages of my study, as well as for serving on my committee. I would also like to thank Dr. Raj Rajkumar for agreeing to serve on my committee and for providing many helpful comments.

During the different stages of my stay at the University of Pittsburgh, I had the chance to meet several noteworthy research colleagues. I would like to thank Libin Dong and Hakan Aydin for their advice, Nevine AbouGhazaleh and Cosmin Rusu for sharing work space and discussing different issues, Ramesh Mishra and Namrata Rastogi for the implementation of my proposed schemes and too many others to name for their kind help. The wonderful time I spent here is always memorable.

Finally, I cannot emphasize enough the support of my parents and my wife to my long education that culminates in this Ph.D. degree. Without their help and sacrifice, none of these achievements would have been possible. While acknowledging that their support was priceless, I dedicate this work to my family.

1.0 INTRODUCTION

The performance of modern computing systems has increased at the expense of drastically increased power consumption. The increased power consumption either reduces the operation time for battery powered systems, such as hand-held mobile systems and remote solar explorers, or generates excessive amount of heat and requires expensive sophisticated packaging and cooling technologies, especially for complex systems that consist of many processing units. The generated heat, if not effectively removed, can also decrease system reliability, since hardware failure rate increases with higher temperature.

To reduce system power consumption, many hardware and software techniques have been proposed and energy aware computing has become an increasing research interest. A system can be turned off when it is idle. However, powering a system up may incur very long latency (e.g., a few minutes [21]). Instead, we can put a system into a power saving state that needs relatively shorter latency for switching back to an active state. For example, Pentium-M processors only take a few cycles for the transition from sleep to active [19].

Moreover, when peak performance is not required, a processing unit can be put into a low performance state that consumes less power. For example, the power consumption of CMOS-based processors is dominated by dynamic power dissipation, which is quadratically related to the supply voltage and linearly related to the processing frequency [13]. Thus, different levels of performance can be delivered with different levels of power consumption by changing processor's frequency and supply voltage, which makes power management at the processor level possible. Based on frequency and/or voltage scaling techniques, energy aware scheduling has been proposed [88], especially for real-time systems, with the goal of using *"the right energy at the right place at the right time"* [36].

Although timeliness is crucial for real-time systems, finishing a real-time application well

ahead of its deadline is not required. When the worst case execution time (WCET) of an application is less than its deadline or an application uses less time than its WCET¹, slack time exists. For example, an automated target recognition algorithm (ATR) detects the region of interest (ROI) in one image frame and compares the ROIs with some templates [75]. When ATR is used in military systems (e.g., missiles), the comparison of ROIs with the templates needs to be done in real-time. However, in most cases, the number of detected ROIs in an image frame may be less than what can be processed. With advanced technologies, slack time in a system can be used by energy management schemes for energy savings.

Many energy aware scheduling algorithms have been proposed for uni-processor systems due to the popularity of mobile systems (i.e., PDAs and cell phones) that generally have one processing unit [6, 62, 67, 72]. However, there is relatively less research focusing on energy management for parallel systems that consist of multiple processing units [28, 54, 86]. Energy management through frequency and/or voltage scaling may change the mapping and scheduling of tasks to processing units, which could cause a violation of the timing constraints even though an application can meet its timing constraints in the worst case scenario when no energy management schemes are employed.

In mission critical real-time systems, such as space-based control systems or life maintenance systems, where a failure may cause catastrophic results, reliability is as important as timeliness. Traditionally, high levels of reliability are achieved through hardware redundancy, such as triple modular redundancy (TMR) [69], where the redundant hardwares consume large amounts of energy. In addition, roll-back recovery and checkpointing have also been proposed to achieve high levels of reliability by exploring slack time as temporal redundancy in the presence of transient faults [45].

Although massive research has been conducted on fault-tolerance and energy aware computing in real-time systems separately, there is relatively less work addressing the combination of energy and reliability management. The complexity of managing them together is partially due to the fact that there is a trade-off between energy consumption and reliability. When more resources are dedicated to fault tolerance schemes for higher levels of system reliability, less resource is left for energy management schemes to save energy. In the domain

¹Real-time tasks usually take 10% to 40% of their WCET [23].

of energy and reliability management, especially for parallel systems, some problems of great practical as well as theoretical interest remain open and challenging.

The goal of this dissertation is to address some open problems in the area of energy aware computing for reliable parallel real-time systems, where both energy efficiency and higher levels of reliability are important. For shared memory parallel real-time systems, we first focus on energy management and propose several energy aware scheduling algorithms. Through simulations, we demonstrate the effectiveness of the proposed algorithms on energy savings. Then, when considering simultaneous management of energy and reliability, we address the interplay between energy consumption and reliability in parallel systems. A few energy efficient fault tolerance schemes are proposed and their performances on energy savings and reliability are analyzed.

In summary, the contributions of this work are as follows:

- Considering all the power consuming components in a system, a simple power model is proposed and its effects on energy management are addressed. Specifically, a minimum *energy efficient frequency* is developed when the system power has a constant component that can be efficiently removed by putting the system into a sleep state. Based on the simple power model, some *theoretical bounds* on the maximum energy savings for parallel systems are developed;
- For shared memory parallel real-time systems, several energy management schemes are proposed and proved to meet the timing constraints of an application while achieving considerable amount of energy savings. More specifically, a scheme of *static power management with parallelism (SPM-P)*, which takes a schedule's parallelism into consideration when carrying out power management, is proposed. Considering the application's run-time behavior, we propose a *shifted/shared slack reclamation (S/SSR)* scheme that shares slack among processing units in a system appropriately for energy savings. *Speculation* schemes are further explored by considering the statistical timing information about an application;
- To efficiently incorporate the overhead of frequency and voltage changes into our energy management algorithms, a *slack reservation* scheme is proposed. The effects of other

practical issues, such as discrete frequency levels and shared memory access contention are also addressed. We find that a few frequency levels are as good as continuous frequency for energy savings obtained by energy management schemes;

- For multiple recovery fault tolerance, we introduce a scheme for computing the optimal numbers of checkpoints to minimize the expected energy consumption for a given reliability goal or to maximize the system reliability for limited energy budget.
- Extending the idea of an optimistic triple modular redundancy (OTMR) scheme [22], an energy efficient optimistic N -modular redundancy (ONMR) scheme is introduced. The optimal frequency settings to minimize the energy consumption of ONMR are analyzed. The energy consumption as well as reliability of ONMR and traditional NMR schemes are compared;
- For fully parallel applications executing on a given number of processing units, the system can be configured with different levels of modular redundancy for fault tolerance, which can tolerate different numbers of faults and consume different amounts of energy. Considering the parallel nature of the slack in parallel systems, an efficient adaptive parallel recovery scheme is proposed and optimal redundant configurations for minimizing energy consumption or for maximizing system reliability are addressed.
- We incorporate the effects of frequency/voltage scaling on the fault rates when studying the trade-off between reliability and energy management. To the best of our knowledge, this is the first work to consider the effects of energy management on reliability.

The organization of this dissertation is as follows. In Chapter 2, the basic definitions are given and the current research status is examined in the area of real-time systems, energy aware computing, fault tolerance and energy efficient fault tolerance. The system models, problem descriptions and research overview are presented in Chapter 3. Chapter 4 reports our research results of energy management for parallel real-time systems and Chapter 5 addresses the energy efficient fault tolerance schemes. Chapter 6 concludes the thesis and Chapter 7 elaborates the research prospects to extend this work in the future.

2.0 BACKGROUND AND RELATED WORK

2.1 REAL-TIME SYSTEMS

The distinguished feature of real-time systems is their timeliness requirements. That is, applications running on such systems have to start execution after their *ready time* (defined as the time at which an application becomes available for execution) and finish the execution correctly before their *deadline*. A real-time application generally consists of a set of tasks and each task has a worst case computation requirement, which can be obtained through profiling or analysis [23, 79].

The tasks in an application may share a common deadline or have individual deadlines. For periodic or frame-based applications, deadlines are generally relative and coincide with the start time of the next period or frame. When tasks in an application have *precedence constraints* (i.e., dependent tasks), a partial order on the execution of tasks is imposed. In addition, tasks can be *preemptive* or *non-preemptive*. While the execution of a preemptive task can be interrupted by another task and resumed later, a non-preemptive task can not be interrupted before its completion.

The scheduling in real-time systems is to decide *which* task is executed *where* (i.e., on *which* processing unit for systems consisting of multiple processing units) at *what* time. A schedule is said to be *feasible* if the precedence constraints, timing constraints (i.e., tasks start after their ready times and finish before their deadlines) as well as any other constraints (e.g., resource constraints that may limit the execution of some tasks on specific processing units) are satisfied.

For parallel real-time applications running on systems consisting of multiple processing units, there are two major strategies to schedule tasks in an application: *partition* and *global*

scheduling [20]. In partition scheduling, each task is assigned to a specific processor and processors can only execute tasks that are assigned to them. This assignment simplifies the complex parallel scheduling problems. In particular, for applications that consist of independent tasks, simple uni-processor scheduling algorithms can be applied to individual processor after partitioning the tasks, and each processor may even have different scheduling algorithms. This strategy is useful for resource restricted systems, where task migration cost is prohibitive or for distributed systems where the cost of migration is very high.

In global scheduling, all tasks are put in a shared global queue and each processor fetches ready tasks from the queue for execution. That is, a task may be executed on any processing unit depending on the dynamic run-time behavior of previous tasks. One good property of global scheduling is that it can automatically balance the actual workload among all processing units, which is especially beneficial for energy management as demonstrated in Chapter 4. The global strategy is useful for shared memory systems where task migration is implicit and the cost is very low.

For tasks with precedence constraints that are represented by directed acyclic graphs (DAG), list scheduling is the standard scheduling technique. Specifically, tasks are put into a ready queue as soon as they become ready and are dispatched to processing units from the front of the queue [20]. The priority assignment for tasks affects *which* task goes *where* as well as the total execution time of an application.

In general, the optimal solution of assigning task priority to get the minimum execution time is NP-hard [20]. Thus, many heuristics have been proposed for that problem, such as *critical path based* heuristics and *cluster based* heuristics [87]. The *longest-task-first (LTF)* heuristic adds tasks to the ready queue in the order of their maximum computation requirements. It has been shown that, for independent tasks, the ratio of the schedule length under LTF heuristics over the one under optimal priority assignment is bounded by a certain constant [65].

For independent periodic tasks running on a single processing unit, the seminal work of Liu and Layland [53] has established the optimality of *earliest deadline first (EDF)* and *rate monotonic scheduling (RMS)* for dynamic and fixed priority policies, respectively. However, they are not optimal for the case of multiple processing units [20]. For restricted harmonic

task sets, where tasks' periods are multiples of each other, several optimal scheduling algorithms have been proposed [43].

For ideal systems, where the allocation can be infinitesimally small, the idea of *generalized processor sharing (GPS)* has been proposed to allocate processor bandwidth fairly to tasks in proportion to utilization requirements [64]. When the allocation time unit is fixed, the *P-fair (proportional fairness)* scheduling for parallel periodic systems was proposed by Baruah et al. in [8], which enforces proportional progress for each task with the allocation error within one time unit. P-fair is optimal in the sense that it can achieve up to 100% system utilization. The P-fair algorithm has been further extended in [2, 3, 4, 9, 33, 61].

2.2 ENERGY AWARE COMPUTING

High system performance is always desired to ensure satisfying the peak computation requirement or meeting the timing constraints for real-time applications. However, maintaining the peak performance all the time in a system may not be a wise decision since the computation requirement in a system generally has big variations and high performance generally implies high power consumption. For example, the average workload for web servers is only 10% to 20% of their peak workload [21]. Thus, it is preferred to tune system performance according to run-time computation requirement while lowering the system power consumption. It is specially useful to extend the operation time of battery operated devices (e.g., cell phones, PDAs and solar explorers) or reduce the operation cost of energy hungry systems (e.g., server farms) and increase their reliabilities.

In order to manage the power consumption in a computing system, many hardware and software techniques have been proposed, such as low energy circuit design and energy aware scheduling. Energy aware computing is to exploit the management schemes that control system performance and energy consumption with the goal of using “*the right energy in the right place at the right time*” [36]. For example, when a system is (and likely going to stay) idle, we can turn it off and remove the power consumption completely [56]. However, powering a system up may take a few minutes [21] which may not be tolerable, especially for real-time applications. Instead, considering the power saving states in the modern processors

[19] and low power memory [24, 73], we can put the system into a power saving state while keeping relatively shorter response time.

Since processors consume substantial energy in most systems, especially in embedded systems, many techniques have been proposed to reduce the energy consumption for processors. For CMOS based processors, the power consumption is dominated by dynamic power dissipation, which is quadratically related to supply voltage and linearly related to processing frequency [13]. In addition, the processing frequency also has an almost linear relation with supply voltage. Therefore, for lower system performance requirements, we can reduce the processing frequency and the corresponding supply voltage to reduce the power consumption cubically.

The idea of trading processing speed for energy savings was first proposed by Weiser et al. in [85], where processor frequency (and corresponding supply voltage) is adjusted using utilization based predictions. Yao et al. described a polynomial-time static off-line scheduling algorithm for independent tasks running with variable frequencies, assuming worst-case execution time [88]. Based on dynamic voltage scaling (DVS) technique, Mossé et al. proposed and analyzed several schemes to dynamically adjust processor speed with slack reclamation [62], where statistical information about task's execution time was used to slow down processor speed evenly and save more energy. In [80], Shin et al. set processor speed at branches according to the ratio of the taken path to the longest path. Kumar et al. predict task execution time based on statistics gathered about execution time of previous instances of the same task [47]. Using stochastic data while taking into consideration actual run-time behavior about tasks, Gruian proposed a two-phase (offline and on-line) algorithm for hard real-time systems with fixed priority assignment for tasks [29]. The best scheme is an adaptive one that takes an aggressive approach while providing safeguards that avoid violating application deadlines [6, 58].

When considering limited voltage/speed levels in real processors, Chandrakasan et al. have shown that, for periodic tasks executing on uniprocessor systems, a few voltage/speed levels are sufficient to achieve almost the same energy savings as infinite voltage/speed levels [17]. Pillai et al. also proposed a set of scheduling algorithms (static and dynamic) for periodic tasks based on EDF/RM scheduling policy, simulations (assuming 4 speed levels)

as well as the prototype implementation show that their algorithms effectively reduce energy consumption from 20% to 40% [68]. AbouGhazaleh et al. have studied the effect of voltage/speed adjustment overhead on choosing the granularity of inserting power management points (PMP) in a program [1].

For systems with several processors running at different fixed speeds and thus with different power profiles, several task assignment and scheduling schemes have been proposed to minimize system energy consumption while still meeting timing constraints of applications (usually represented by directed acyclic graphs) [30, 44, 86]. When variable voltage processors are used, for fixed task sets and predictable execution times, static power management (SPM) can be accomplished by deciding beforehand the best voltage/speed for each processor [28]. For periodic task graphs and aperiodic tasks in distributed systems with a given static schedule for periodic tasks and hard aperiodic tasks, Luo et al. proposed a static optimization algorithm by shifting the static schedule to re-distribute static slack according to the average slack ratio on each processor element. This reduces energy consumption and response time for soft aperiodic tasks at run time [54]. They improved the static optimization by using critical path analysis and task execution order refinement to get the maximal static slow down factor for each task [55]. In [55], a dynamic voltage scaling scheme is also proposed.

For tasks with precedence constraints and a given task assignment, Gruian et al. proposed a priority based energy sensitive list scheduling heuristic to determine the amount of time allocated to each task, considering energy consumption and critical path timing requirement in the priority function [31]. In [92], Zhang et al. proposed a mapping heuristic for fixed task graphs to maximize the opportunities for voltage scaling algorithms, where the voltage scaling problem was formulated as an integer programming (IP) problem.

2.3 FAULT TOLERANCE

During the execution of an application, a fault may occur due to various reasons, such as hardware failures, software errors and electro-magnetic effects. Thus, fault tolerance is an inherent requirement of real-time systems when correct results are needed even in the

presence of faults [40].

A fault is a defect, imperfection, or flaw of a particular hardware/software component. An error is the manifestation of a fault, and it is a property of the state of a system. A failure occurs when one system performs its functions incorrectly as a result of an error. According to their temporal behaviors, faults can be divided into *permanent faults*, which may result in total failure of a processing unit, and *transient faults*, which affect a processing unit temporarily and disappear after a relatively short period. Between these two categories, the *intermittent faults* are the most troublesome ones since they appear and disappear randomly in time.

To tolerate faults, the first step is *fault detection*. Faults can be discovered through self-coding programming or by comparing results from replicated execution [40]. Fault tolerance schemes generally rely on some form of *redundancy*. This means that extra resources (e.g., hardware, software or time) are needed in addition to what is required to perform the normal operation. Hardware redundancy, software redundancy, time redundancy and information redundancy are the main types of redundancy used in contemporary reliable systems.

Permanent faults are generally tolerated by hardware redundancy, which is also known as modular redundancy (MR), where cloned tasks are running concurrently on multiple processing units [69]. To further prevent software errors, *N-version programming* [5], a software redundancy technique, can be combined with modular redundancy to execute several different versions of the same task on multiple processing units. The well known software redundancy technique is the *recovery block* approach proposed by Randell [74].

For real-time tasks to tolerate transient faults, temporal redundancy can be explored. The rollback recovery scheme restores the system state to a previous safe state (e.g., start a system from scratch and re-load a task) and repeats the computation [45]. However, it is inefficient to repeat the computation from scratch, especially for long execution of large tasks. Checkpointing techniques have been proposed to efficiently use temporal redundancy by periodically saving the important system information to stable storage and run an error-detection routine simultaneously [10]. If errors are detected, the correct state in previous checkpoint is restored and the computation within these two checkpoints is repeated. The optimal checkpoint interval is explored in [49]. Instead of rollback and re-executing a faulty

section immediately, roll-forward schemes are proposed, where the processing units that detected faults continue execution while an additional processing unit is used to locate the fault and to synchronize the correct state among processing units later [70].

In parallel systems, many scheduling algorithms for fault tolerance have been proposed [50, 51, 57, 63]. The primary/backup recovery scheme explicitly schedules a backup task for every task using space and/or temporal redundancy [25]. Instead of executing the same computation, a recovery block with identical or similar functionality may be substituted. For the case where several recovery blocks exist, the recovery blocks are run one after another as long as the error persists. Obviously, the functionality and the number of recovery blocks are limited by the amount of temporal redundancy available [5].

2.4 ENERGY EFFICIENT FAULT TOLERANCE

Fault-tolerance schemes through redundancy have energy implications and different techniques have different effects on energy consumption.

For a set of independent periodic tasks, using the primary/backup recovery scheme, Unsal et al. proposed an energy-aware software-based fault tolerance scheme, which postpones as much as possible the execution of backup tasks to minimize the overlap of primary and backup execution and thus to minimize energy consumption [84]. Checkpointing was explored to tolerate a fixed number of faults while minimizing the energy consumption for a periodic task set [91], and the work was further extended in [90] by considering faults within checkpoints.

For a single task application to tolerate one fault, the optimal number of checkpoints, evenly or unevenly distributed, to achieve minimal energy consumption was explored in [59]. Assuming a Poisson fault model, Zhang *et al.* proposed an adaptive checkpointing scheme that dynamically adjusts checkpoint intervals to tolerate a fixed number of faults [89]. Elnozahy *et al.* proposed an *Optimistic TMR* scheme to reduce the energy consumption for traditional TMR systems by allowing one processing unit to slow down provided that it can catch up and finish the computation before the application deadline if the other two processing units do encounter faults [22].

3.0 SYSTEM MODELS AND PROBLEM DESCRIPTION

In this dissertation, we focus on the energy and reliability management for parallel real-time systems. Section 3.1 will first present the application and system models. Then the power model and its effects on energy management are discussed in Section 3.2. Section 3.3 presents the fault and recovery models and Section 3.4 describes the problems and summarizes our research results.

3.1 APPLICATION AND SYSTEM MODELS

We concentrate on *frame-based* real-time applications, which consist of a set of *independent* or *dependent* real-time tasks [50]. With a common *deadline*, D , which is also the size of an application frame, all tasks within an application need to finish their execution before D . Although we focus on the case where all tasks share a common deadline, it is easy to extend some of our main results to the case where each task has its individual deadline as mentioned in Chapter 7.

The task set of an application is defined as $\Gamma = \{T_1, \dots, T_n\}$ and each task T_i is represented by its two attributes, c'_i and a'_i , which are the *maximum* and *average* computation requirement (in terms of number of cycles). Given that variable frequency/voltage processors are considered, the number of cycles needed to execute a task may also depend on its processing frequency when the memory access time is fixed [79]. However, when the processor cache has a reasonable size, c'_i has been shown to have very small variations with different processing frequencies [59]. In this dissertation, for simplicity, we assume that c'_i and a'_i are constants and correspond to the worst case and average number of cycles needed

to execute task T_i at the maximum processor frequency f_{max} , respectively. Notice that, this is a conservative model. When memory effects are considered, the number of cycles needed to execute a task decreases when executing at lower frequencies since memory access time is fixed [79].

For ease of presentation, we further define c_i and a_i as the worst case execution time (WCET) and the average case execution time (ACET) of task T_i at frequency f_{max} , respectively. With c'_i and a'_i assumed constants, the execution time of a task T_i increases proportionally when processor frequency is reduced. For example, with half of the maximum processor frequency, $\frac{f_{max}}{2}$, the WCET of task T_i is assumed to be $2 \cdot c_i$.

Based on whether all tasks in an application are executed during any execution of the application, we consider two different application models: the *AND-model* and the *AND/OR-model*. For AND-model applications, all tasks are executed during any execution instance of an application. However, only a subset of tasks will be executed during one execution instance of an AND/OR-model application.

3.1.1 AND Application Model

If an application consists of independent tasks, there is no constraint on the order of tasks to be executed. However, the precedence constraints of dependent tasks impose a partial order on the execution of tasks. In this work, a directed acyclic graph (DAG) G is used to represent the precedence constraints among tasks [87].

A DAG that represents the precedence constraints of an AND-model application is defined as $G = \{\tilde{V}, \tilde{E}\}$, where each vertex $v_i \in \tilde{V}$ corresponds to a task $T_i \in \Gamma$ and $\tilde{E} \subseteq \tilde{V} \times \tilde{V}$ represents the data dependencies between tasks. For the case of independent tasks, there is no dependency between tasks and \tilde{E} is empty. There is an edge $e_{ij} :: v_i \rightarrow v_j \in \tilde{E}$ if task T_i is an immediate predecessor of task T_j , which means that T_j needs the output data from T_i as its input data. In other words, T_j becomes *ready* for execution only after T_i finishes its execution.

Each vertex v_i in a DAG G is labeled by the corresponding task T_i and its two attributes, c_i and a_i , which are the maximum and average case execution time, respectively.

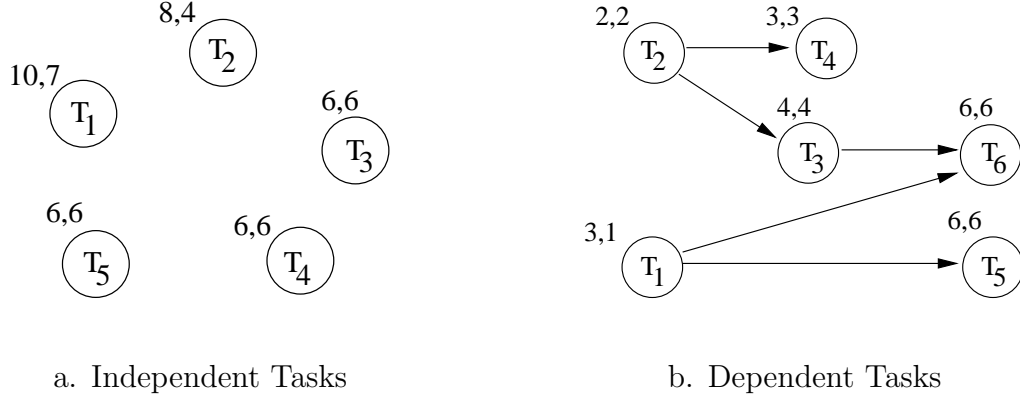


Figure 3.1: AND-model application examples.

For independent tasks as illustrated in Figure 3.1a, all tasks are ready to be executed at the beginning of a frame. When precedence constraints exist between tasks, only *root tasks*, which are defined as tasks that have no predecessor (e.g., tasks T_1 and T_2 in Figure 3.1b), are ready at the very beginning of a frame. For tasks with more than one predecessors, they are ready to be executed only after all their predecessors finish execution. For example, task T_6 is ready only after T_1 and T_3 finish their execution (see Figure 3.1b).

Tasks may be executed in parallel if they are ready and there are available processing units. For example, for a system consisting of three processing units, the independent tasks T_1 , T_2 and T_3 (Figure 3.1a) can be executed in parallel on three processing units. However, for the dependent tasks (Figure 3.1b), only the root tasks T_1 and T_2 can be executed at the very beginning since other tasks are not ready.

3.1.2 AND/OR Application Model

Although the AND-model is sufficient to represent applications that have only data dependencies, it cannot describe control dependencies. The OR structures exist in the control flow of most practical applications, where the structure of *if-then-else* is employed and the execution of sub-paths depends on the results of previous tasks. In some applications, the probability of a path to be executed may also be known a priori or can be obtained from profiling. To represent all these features, for applications where a task is ready when one *or more* of its predecessors finish execution, and one *or more* of its successors become ready

after the task finishes execution, we extend the AND/OR-model developed in [26].

In addition to computation nodes as in the AND-model, to represent control flow and parallelism in an application, we define two other types of nodes: *AND* and *OR* synchronization nodes. That is, vertices in a DAG that represents an AND/OR-model application will include task nodes as well as synchronization nodes.

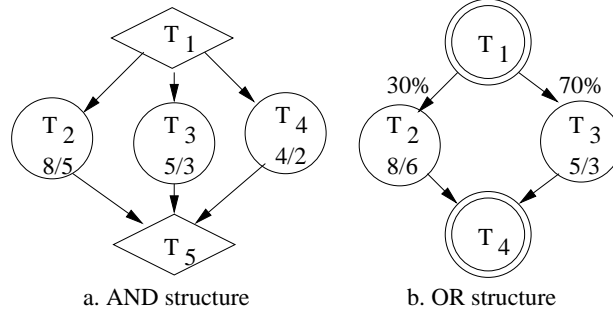


Figure 3.2: AND/OR structures

An AND node is represented by a *diamond*, which depends on all its predecessors (that is, it can start execution only after all its predecessors finish execution) and all its successors depend on it (that is, all its successors can execute only after it finishes execution). It is used to explore the parallelism in parallel applications as shown in Figure 3.2a, where tasks T_2 , T_3 and T_4 can be executed in parallel after the AND node T_1 finishes execution.

An OR node is represented by a *dual-circle*, which depends on only one of its predecessors (that is, it is ready for execution as soon as any one of its predecessors finishes execution) and only one of its successors depends on it (that is, exactly one of its successors is executed after it finishes execution). It is used to explore the different execution paths in applications as shown in Figure 3.2b. To represent the probability of taking each branch after an OR synchronization node, a number is associated with each successor of an OR node. For example, in Figure 3.2b, the path along task T_2 has the probability of 30% to be executed, while the path along task T_3 has the probability of 70% to be executed.

As in the AND-model, a computation node T_i is represented by a *circle* and labeled by its two attributes c_i and a_i . The AND/OR nodes are considered as *dummy* tasks with the worst case execution time being 0 (it is easy to transform a synchronization node with non-zero

computation requirement to a synchronization node and a computation node).

For simplicity, we only consider the case where an OR node cannot be processed concurrently with other paths. In other words, all processing units will synchronize at every OR node. We define a *segment* as a set of tasks that are separated by two adjacent OR nodes. There will be several segments, one for each branch, between two OR nodes.

Notice that there is no back-edge in DAGs. For a loop in an application, given the maximum number of iterations, q , and the corresponding probabilities to execute a specific number of iterations, we can either treat the whole loop as a single task that has a specific c_i and a_i computed as the worst and average execution time needed for the entire loop, respectively, or we can expand the loop into several tasks as described next.

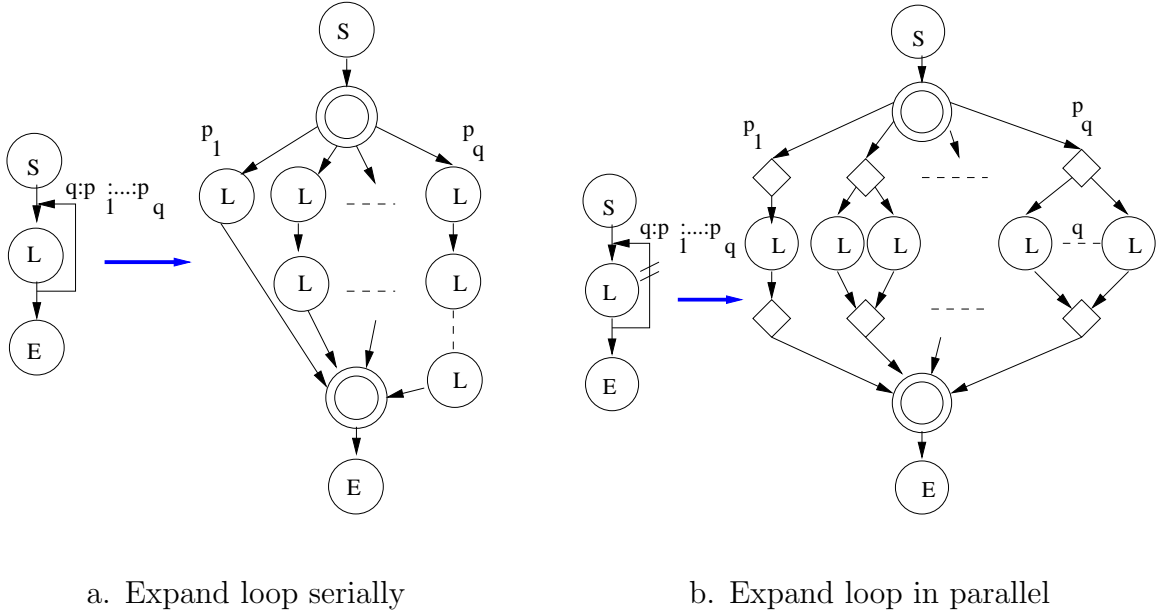


Figure 3.3: Loop expansion in AND/OR-model applications.

Based on whether there is dependence between iterations, there are two ways to expand a loop. If there is dependence between iterations (i.e., iteration $i + 1$ depends on iteration i , $i = 1, \dots, q - 1$), the loop can only be expanded serially (Figure 3.3a), where each branch represents the case of a specific number of iterations to be executed. If there is no dependence between iterations (represented by two parallel lines across the back edge of a loop), the loop can be expanded in parallel as shown in Figure 3.3b. In Figure 3.3, the probability, p_j , of

having j iterations is associated with the corresponding successor of the OR node.

An example of AND/OR-model applications is shown in Figure 3.4. Notice that the AND-model is a special case of the AND/OR-model. However, due to its relative simplicity, we present the AND-model separately for ease of discussion later. As shown in Chapter 4, it is easier to present the energy management schemes for AND-model applications and then extend them appropriately to AND/OR-model applications.

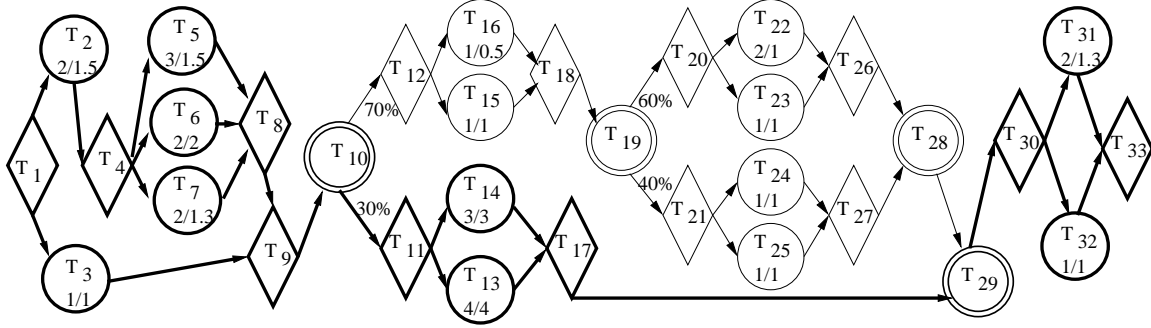


Figure 3.4: An example of AND/OR-model applications.

3.1.3 System Models

We consider *shared-memory* parallel systems in this work. In shared-memory systems, the inter-processor communication is implicit by writing/reading the same memory location. Unless specified otherwise, it is assumed that the inter-processor communication cost is negligible. That is, if dependent tasks run on different processing units in a shared-memory system, the data generated by predecessor tasks is available to their successor tasks immediately after they finish execution.

We further assume that the systems considered are homogeneous. That is, the processing units in such systems are *identical*. Tasks in a parallel application can be executed by any one of the processing units, which have the same processing capability and power characteristics. Task migration among processing units is implicit and is assumed to be free.

We further assume that the performance and power consumption of the systems considered are manageable. That is, the hardware subsystems have the capability of changing their

working states. For example, RDRAM, an energy efficient memory, can be put into different power saving states that have different response times [73]. Moreover, variable voltage processors, such as Transmeta [37], Intel Xscale [34] and AMD K6+ [68], can change their power consumption by adjusting processing frequency and supply voltage within certain ranges. For simplicity, unless specified otherwise, we use normalized processing frequency and supply voltage. The maximum processing frequency is assumed to be $f_{max} = 1$ with the corresponding supply voltage $V_{max} = 1$. The minimum processing frequency is f_{min} with supply voltage V_{min} . The system power model is further discussed in the next section.

3.2 POWER MODEL AND ITS EFFECTS ON ENERGY MANAGEMENT

In a computing system, the power is consumed mainly by processors, memory, disks, network interface, clock generator and underlying circuits. While the power consumption of some components can be controlled in a fine granularity (e.g., processors and memory), other components (e.g., disks and underlying circuits) can only be turned on or off. In this section, we first address the dynamic power model for CMOS based processors, then we propose a simple power model considering all power consuming components in a computing system. The effects of the simple power model on energy management are also discussed.

3.2.1 Dynamic Power for CMOS Based Processors

The power consumption for CMOS based processors is dominated by dynamic power dissipation P_d , which is given by [13, 18]:

$$P_d = C_{ef} \cdot V_{dd}^2 \cdot f \quad (3.1)$$

where C_{ef} is the effective switch capacitance, V_{dd} is the supply voltage and f is the processor clock frequency. That is, P_d is quadratically related to supply voltage and linearly related to processor frequency. Since the circuit delay is also determined by supply voltage, there is

an almost linear relation between processor frequency f and supply voltage V_{dd} as shown in the following equation:

$$f = k \cdot \frac{(V_{dd} - V_t)^2}{V_{dd}} \quad (3.2)$$

where k is a constant and V_t is the threshold voltage [13, 18].

Voltage scaling technique reduces supply voltage for lower processor frequencies when non-peak performance is required [66]. This reduces processor dynamic power consumption cubically at the expense of linearly decreasing processor frequency and linearly increasing the execution time of an application. In this work, we use frequency changes to stand for changing both frequency and supply voltage simultaneously.

3.2.2 A Simple Power Model

Although dynamic power dominates processor power dissipation, static leakage power increases very fast with technology advancements and will probably be comparable to dynamic power dissipation within a few technology generations, especially with *sub-micron* technology. For example, the processor static leakage power for $1\mu\text{m}$ technology was 0.01% of total power, but is approaching 10% for $0.1\mu\text{m}$ technology [83].

In addition to processors, there are other components that consume power in a system. Since energy is defined as the integral of power over time, the energy consumption E of a system to execute an application at power level P for time t is $E = \int P(t)dt$. While scaling down frequency and voltage can save dynamic energy, it may not save the total system energy to execute a specific application since the application will take more time and consume more energy in other components. Thus, it is necessary to incorporate the power consumed by other components into the power model when applying frequency and voltage scaling techniques for energy savings.

The static leakage power may be removed by putting processors into a sleep state when a system is idle. For example, the maximum active power for Intel Pentium-M processors is around $25W$ and minimum active power is around $4W$. However, they only consume around $0.5W$ in sleep state and need a few cycles for returning back to active working state [19].

Thus, when there is no computation requirement, we can put a system into a power saving sleep state, from which a system can quickly (e.g., in a few cycles) respond to computation requirement. Moreover, for the maximum power savings, we can also power off a processing unit when it *is* and *will possibly remain* idle for a while and power it up when needed. However, turning on/off a system incurs large time and energy overheads [11].

To incorporate all power consuming components in a processing unit and keep the power model simple, we assume that a processing unit has three different states: *active*, *sleep* and *off*. The *active state* is defined as having computation in progress. Depending on different processing frequencies, all static power and different amounts of dynamic power are consumed in the active state. The *sleep state* is a power saving state that removes all dynamic power and most of the static power. Processing units in sleep state can act quickly (e.g., in a few cycles) to new computation requirement. The transition timing overhead from sleep state to active state is assumed to be negligible. A processing unit is assumed to consume no power in the *off state*. For processing units that support automatic powering on (e.g., wake-on-LAN), the corresponding component (e.g., the network interface) needs to be in a working state while a processing unit is off. However, the power consumption to maintain that component is small and assumed to be negligible.

Thus, the power consumption of a processing unit at frequency f can be modeled as:

$$P(f) = P_s + \hbar(P_{ind} + P_d) = P_s + \hbar(P_{ind} + C_{ef}f^m) \quad (3.3)$$

where P_s is the sleep power; P_{ind} and P_d are frequency-independent and frequency-dependent active powers, respectively. \hbar equals 1 if the system is active and 0 if the system is in the sleep state. C_{ef} and m (> 2) are system dependent constants and f is the processing frequency.

The sleep power P_s includes (but is not limited to) the power to keep basic circuits active, to keep the clock generator running, and to maintain processor and memory in a sleep mode [19, 48]. Active power is further divided into two parts: the *frequency-independent active power* and the *frequency-dependent active power*. Frequency-independent active power consists of part of the memory and processor power as well as any power that can be efficiently

removed by putting the processing units to sleep and is independent of the processing frequency and supply voltage. Frequency-dependent active power includes processor's dynamic power and any power that depends on the processing frequency and supply voltage [13, 82].

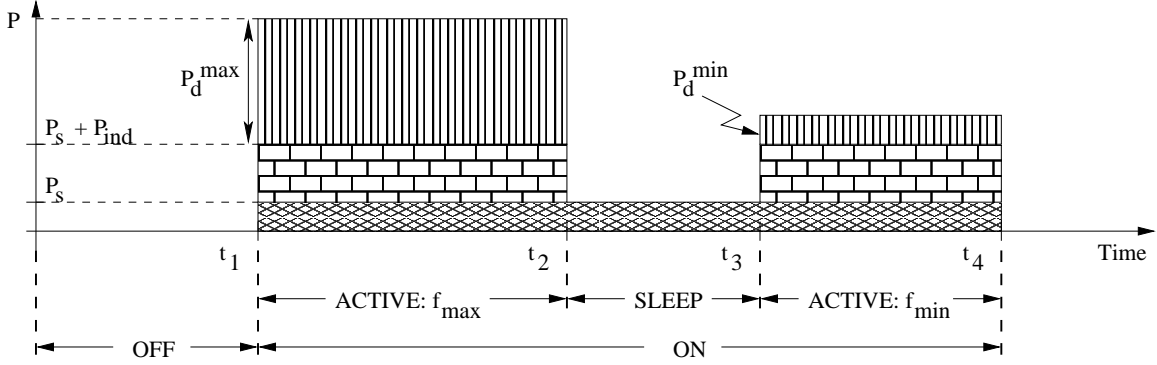


Figure 3.5: The simple power model: power consumption in different states.

The simple power model is further illustrated in Figure 3.5, where the power consumption for different states in a processing unit is shown. In the figure, the Y-axis is the total power in a processing unit and the X-axis represents time. Initially, the processing unit is off and consumes no power. From time t_1 to time t_2 , it is powered on and runs at frequency f_{max} . Thus, the maximum power ($P_s + P_{ind} + P_d^{max}$) is consumed. After t_2 , the processing unit is put to sleep since there is no computation and only sleep power P_s is consumed. During period of (t_3, t_4) , it is active again and runs at f_{min} . While the frequency-dependent active power is reduced to P_d^{min} , P_s and P_{ind} remain the same during active state.

Notice that, for simplicity, the transition from the off state to the on state is not shown in the figure. However, it takes considerable amount of time and consumes a certain level of power to turn a processing unit on. For example, up to one minute has been reported to turn on a PC running Windows [11].

To validate the effectiveness of our power model when modeling the power consumption in a system, we analyze the power consumption for Intel XScale processors and determine the parameters in our power model [34]. The frequencies and corresponding power consumption for Intel XScale processors are shown in Table 3.1. Notice that, part of the power for each frequency level is the sleep power P_s and frequency independent active power P_{ind} . Consid-

ering the power characteristics of other components in a systems, their power consumption contributes mostly to P_s and P_{ind} . However, due to the lack of such real measured power numbers, we validate our power model considering processors only.

Table 3.1: Power consumption at different frequencies for Intel XScale processors.

$f(GHz)$	1	0.8	0.6	0.4	0.15
$P(W)$	1.6	0.9	0.42	0.14	0.05

Using curve fit nonlinear regression techniques [35], we fit the powers and frequencies of Intel XScale processors with our simple power model. Notice that, for different values of P_s and P_{ind} , after fitting the frequency and power number with our power model, we will get different standard errors and correlation coefficients. The smaller standard errors and larger correlation coefficients indicate better fit between the model and the power and frequency pairs. Figure 3.6 shows the standard errors and correlation coefficients for different vaules of $P_s + P_{ind}$ when applying nonlinear regression curve fitting of the powers and frequencies with our power model.

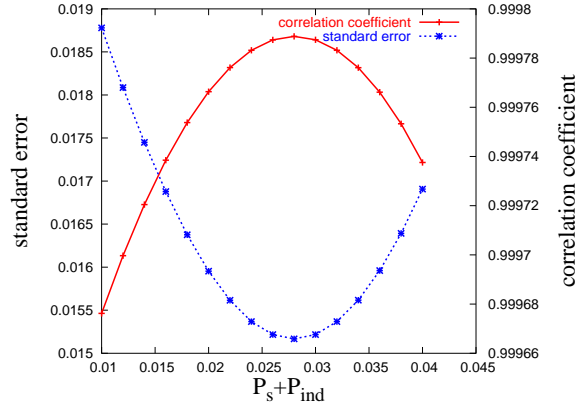


Figure 3.6: The standard errors and correlation coefficients for different values of $P_s + P_{ind}$ when fitting the powers and frequencies of Intel XScale into our power model.

From the figure, we can see that when $P_s + P_{ind} = 0.028W$, we get the best fit with the minimum standard error 0.01517 and the maximum correlation coefficient 0.99979. For all the values of $P_s + P_{ind}$ from 0.01 to 0.04 when fitting the powers and frequencies with our

power model, the coefficient C_{ef} has a range from 1.592 to 1.566 and the exponent m has a range from 2.644 to 2.767, respectively. For the case of best fit (i.e., $P_s + P_{ind} = 0.028W$), there are $C_{ef} = 1.577$ and $m = 2.717$. Notice that, if the power is modeled as a 3^{rd} degree polynomial relation with frequency, we can get $P = 0.121 - 0.7664f + 1.8978f^2 + 0.3475f^3$ with the standard error as 0.000778 and correlation coefficient as 0.999999. Considering the small difference (0.02% on correlation coefficient) between these two models, we will adhere our simple power model in the thesis.

3.2.3 Effects of Power Model on Voltage Scaling

As we mentioned in Section 3.2.1, scaling down system processing frequency and supply voltage can save frequency-dependent active energy, but it takes longer to execute an application and consequently it will consume more sleep energy and frequency-independent active energy. Therefore, there should be an energy efficient frequency f_{ee} to minimize system energy consumption [22, 24, 38, 76].

For ease of discussion, the maximum frequency-dependent active power is assumed to be $P_d^{max} = C_{ef}f_{max}^m = 1$. Moreover, P_s and P_{ind} are assumed to be αP_d^{max} and βP_d^{max} , respectively. An application that needs L time units at $f_{max} = 1$ will take $\frac{L}{f}$ time units when running at frequency f . Therefore, the system energy consumption to execute the application at frequency f is:

$$E = (P_s + P_{ind} + C_{ef}f^m) \frac{L}{f} \quad (3.4)$$

Differentiating Equation (3.4) with respect to f and setting the result to zero, we find that E is minimized when $f = \sqrt[m]{\frac{\alpha+\beta}{m-1}}$, which is defined as the *energy-efficient* frequency f_{ee} . Notice that an application cannot run faster than the maximum frequency f_{max} . If $f_{ee} > f_{max}$, that is, $\alpha + \beta > m - 1$, all applications would run at f_{max} to minimize their energy consumption and no voltage scaling is necessary.

For systems that do not support turning on/off processing units or when the overhead of turning processing units on/off is prohibitive [11], the system is assumed to be always on and the sleep power P_s is not manageable (i.e., always consumed). In this case, the energy efficient frequency can be easily found as $f_{ee} = \sqrt[m]{\frac{\beta}{m-1}}$.

Given that f_{min} is the minimum supported processing frequency, we define the minimum energy efficient frequency as $f_{low} = \max\{f_{min}, f_{ee}\}$ and $\kappa = \frac{f_{low}}{f_{max}}$. That is, we may be forced to run at a frequency higher than f_{ee} to meet the deadline of an application or to comply with the lowest frequency limitation, but we should never run at a frequency below f_{ee} , since doing so consumes more energy. Unless specified otherwise, we assume that $f_{ee} \geq f_{min}$ in this work. That is, $f_{ee} = \kappa$.

Thus, **when the system power has a constant component that can be efficiently removed by putting the system into a sleep state when the system is idle, its effect on voltage scaling for energy saving is equivalent to imposing a lower bound on the processing frequency.**

3.3 FAULT AND RECOVERY MODELS

Since transient and intermittent faults occur much more frequently than permanent faults [16], in this work, we consider only transient faults and focus on exploring slack time in a system as temporal redundancy for increasing reliability.

3.3.1 Fault Models

Two different fault models will be considered: a *deterministic* model and a *probabilistic* model. For the deterministic model, we explore recovery schemes to tolerate a fixed number (e.g., k) of faults during the execution of an application. That is, the system can recover from at most k faults and the application will finish its execution in a correct and timely fashion. If more than k faults occur, then the application will fail.

For the probabilistic model, as usual, faults are assumed to follow a Poisson distribution with an average fault arrival rate λ . That is, for an application with execution time t , if there is no recovery, the probability of failure (i.e., having fault(s) during its execution) is:

$$\rho = 1 - e^{-\lambda \cdot t} \quad (3.5)$$

The average fault arrival rate λ may change under different environments. The effects of voltage scaling on λ will be further addressed in Chapter 5.

3.3.2 Rollback Recovery and Checkpoints

Faults are assumed to be detected through replicating the execution of an application on multiple processing units and comparing their results. If results are the same, there is no fault. That is, we do not consider correlated faults that result in the same error on every processing unit. If the results are not the same, faults are detected. If the faults can not be masked through majority voting in a modular redundant system, recovery is needed.

In this work, we consider rollback recovery [45] and explore checkpointing techniques for efficient usage of temporal redundancy. At a checkpoint, the important system information is saved to stable storage and a fault-detection process is executed [10]. If a fault is detected, the correct state stored during the previous checkpoint is restored and the computation within these two checkpoints is repeated [45].

The time overhead of one checkpoint is assumed to be r , which includes the time for saving system states, running the fault detection process and restoring previous correct states if needed. With different checkpointing techniques, the overhead of taking a checkpoint has large variations [71]. We will explore the effects of checkpointing overhead on our management schemes in Chapter 4.

3.4 PROBLEM DESCRIPTION AND RESEARCH OVERVIEW

For a parallel frame-based real-time application, which consists of independent or dependent tasks, we first consider the problem of minimizing energy consumption while guaranteeing the timing constraints of the application when it is executed on a homogeneous shared-memory system that has multiple processing units. When system reliability is also a major concern, the problem becomes exploring the optimal fault tolerance scheme that either minimizes energy consumption for a given system reliability target or maximizes system reliability within limited energy budget while the application's timing constraints are still satisfied.

Figure 3.7 shows the research scope and summarizes the solutions proposed in this work. We first consider the energy minimization problem for applications being executed in a

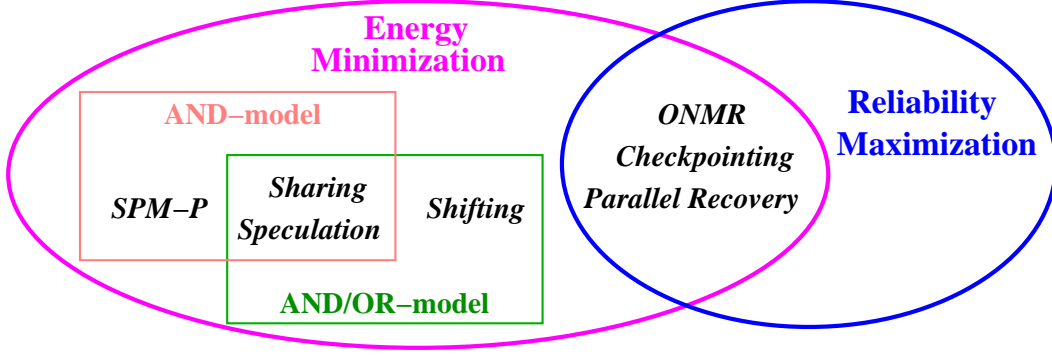


Figure 3.7: Summary of solutions and research overview.

fault-free environment. Then, focusing on tolerating transient faults, energy efficient fault tolerance schemes are proposed.

As the first part of this work, static power management is addressed. It is well known that the uniform static power management (U-SPM), which evenly scales down the processing frequency when executing an application by proportionally distributing static slack over the schedule, is optimal for uniprocessor systems. However, we found that U-SPM is not optimal for parallel systems where the degree of parallelism in a schedule, defined as the number of processing units that have tasks for execution at a given moment, may vary. With the observation that more energy can be saved by allocating more slack to schedule sections with higher levels of parallelism, a *static power management scheme with parallelism* (*SPM-P*) is proposed (see Section 4.2.3).

When the dynamic behavior of tasks is considered, energy management for parallel systems becomes more complicated. This is because the mapping and/or scheduling of tasks to processing units may change and cause timing constraint violations due to the variations of tasks' execution time. Based on a global scheduling strategy, we first propose a *slack sharing* scheme for AND-model applications to reclaim dynamic slack for energy savings (Section 4.3.2). Considering the slack time resulting from executing different paths in AND/OR-model applications, an extended *slack shifting/sharing* scheme is proposed (Section 4.3.3). Moreover, *speculation* schemes, which use the statistical timing information of applications, are proposed with the intention of saving more energy by reducing the number

of voltage/frequency changes and the related overheads (Section 4.4).

Taking reliability into consideration, *checkpointing* techniques are first explored to efficiently use the slack time in a system when roll-back recovery schemes are used for fault tolerance. We propose the concept of *level of pessimism*, which denotes the number of faults expected to occur in the analysis. Based on this concept, we explore the optimal number of checkpoints to minimize expected energy consumption with a given reliability goal. We also devise the optimal number of checkpoints to maximize system reliability with limited energy budget (Section 5.1).

Extending the idea of an optimistic triple modular redundancy (OTMR) scheme [22], we consider an energy efficient *optimistic modular redundancy (ONMR)* scheme. For that, we assume that faults are rare and some processing units in a modular redundant system are turned off or scaled down for energy savings, provided that they can catch up and finish the computation before the application’s deadline if other processing units do encounter faults. The optimal frequency setting for the processing units in an ONMR system is explored (Section 5.2).

For an application being executed on a system that consists of a certain number of processing units, an efficient *parallel recovery* scheme is proposed and the optimal redundant configuration problem is addressed. Here, the redundant configuration is defined by the level of modular redundancy employed, the number of processing units used, the frequency of the active processing units and the number of backup time units needed. Furthermore, we discuss a framework for finding the optimal redundant configuration to minimize the expected energy consumption for a given reliability target as well as to maximize system reliability for a given energy budget (Section 5.3).

Finally, the trade-off between energy consumption and system reliability is addressed while considering the effects of energy management on fault rates. We illustrate that ignoring the effects of voltage scaling on fault rates (i.e., fault rates increase when voltage is reduced) may lead to unsatisfied system reliability when voltage scaling is used for energy savings (Section 5.4).

4.0 PARALLEL ENERGY AWARE SCHEDULING

In this chapter, we focus on exploring slack time for energy saving for frame-based parallel real-time applications that are executed on systems consisting of multiple processing units. We will first illustrate the importance of task priority assignment on meeting an application's timing constraints in parallel systems and explain why we choose fixed-priority list scheduling in this work (Section 4.1). Then we consider energy management schemes that explore static as well as dynamic slack in Section 4.2 and Section 4.3, respectively. The evaluations of the proposed energy management schemes are presented in Section 4.6. Section 4.7 develops some theoretical bounds on the energy savings that can be obtained through energy management and Section 4.8 summarizes this chapter.

As a first step, when presenting the energy management schemes, we focus on frequency-dependent active power $P_d = C_{ef}f^m$ (see Section 3.2) and ignore sleep power and frequency-independent active power (i.e., we assume that $P_s = P_{ind} = 0$). However, for systems where P_s and P_{ind} are significant, the energy management schemes can be easily adapted by imposing a minimal energy-efficient frequency limitation as discussed in Section 3.2. The effects of P_s and P_{ind} on energy management are further evaluated in Section 4.6.6. Moreover, unless specified otherwise, we assume that $m = 3$.

4.1 SCHEDULING IN PARALLEL REAL-TIME SYSTEMS

We start by illustrating the importance of execution order on meeting the timing constraints of an application. Although there is no power management addressed in this section, the results presented here will be used by power management schemes in the following sections.

There are two major strategies for mapping tasks in parallel applications on multiple processing units: *global* and *partition* scheduling [20]. In global scheduling, all ready tasks are put into a global queue and each idle processing unit fetches from the queue the task with the highest priority for execution. In the partition scheduling strategy, ready tasks are mapped to individual processing unit and each processing unit fetches the highest priority task from its own queue for execution. Although the simple uni-processor energy management schemes [62] can be applied directly on each processing unit in partition scheduling, global scheduling has the merit of automatic balancing the actual workload among processing units [20], which implies the possibility of uniformly scaling down all processing units for more energy savings. Thus, in this work, we focus on global scheduling. Moreover, we consider only *non-preemptive* scheduling, in which a scheduled task run to completion before a processing unit fetches another task for execution.

List scheduling is a well-known technique to schedule dependent tasks represented by DAGs on multiple processing units [65, 87]. It puts tasks into a ready queue as soon as they become ready and dispatches the task at the head of the ready queue to an idle processing unit. Notice that, the case of independent tasks can be considered as a special case of dependent tasks, where there is no precedence constraint and all tasks are ready at the beginning of a frame.

4.1.1 Earliest Ready Longest Task First Heuristic (ER-LTF)

We can see that the priority assignment of tasks affects *which* task is executed on *which* processing unit, the workload of each processing unit, and the schedule length (defined as the total time needed to finish executing all tasks). When tasks are ready at the same time, in general, the problem of finding the optimal solution of assigning task priority to minimize schedule length is NP-hard [20]. Moreover, as we show in [95], the optimal priority assignment that minimizes schedule length of an application may not lead to the minimum energy consumption due the application’s run-time behaviors.

In this work, we consider the *longest task first (LTF)* heuristic when determining the priority of tasks that are ready at the same time. In general, we expect that longer tasks

(based on tasks' WCET) generate more dynamic slack at run-time, which can be used by future tasks, and thus more energy savings could be obtained. It also has been shown that, for independent tasks, the ratio of the schedule length under LTF heuristic over the one under optimal priority assignment is bounded by a certain constant [65]. In addition, LTF heuristic is simple and easy to be implemented.

Due to the precedence constraints among tasks, the ready times of tasks may be different. In order to obtain a total order on tasks' priorities, we extend LTF and use the *earliest ready longest task first (ER-LTF)* heuristic to assign tasks' priorities. That is, tasks with earlier ready time have higher priorities. If tasks' ready time is the same, longer tasks have higher priorities. If there is a tie, it can be broken arbitrarily (e.g., by task ID). Therefore, the priority of a task T_i is defined as a tuple (rt_i, c_i) , where rt_i is the ready time of task T_i and c_i is T_i 's WCET.

A *schedule* is a list of events that specify the mapping, as well as start and end times of tasks for an application running on a specific system. A schedule is *feasible* if it meets all the precedence and timing constraints of an application. An application is *schedulable* if there exists a priority assignment such that there is a feasible schedule even in the worst case scenario. Note that the ER-LTF heuristic is not optimal. That is, an application may not be able to meet all its precedence and timing constraints under ER-LTF heuristic even if the application is schedulable. In this work, for the applications we considered, a feasible schedule is assumed to exist under ER-LTF priority assignment heuristic in the worst case scenario.

However, it is worth to point out that the energy management schemes proposed later in this chapter do not depend on specific priority assignment heuristics. For any priority assignment algorithm, if it can generate a feasible schedule for an application in the worst case scenario, our energy management schemes can be applied for energy saving and still guarantee that the application's precedence and timing constraints are met.

4.1.2 Canonical Schedules

Under a certain scheduling policy, if all tasks use their worst case execution time and meet the precedence and timing constraints during an execution of an application, the execution is defined as *canonical execution*. A schedule that has the same mapping, as well as start and end times for all tasks as those in a canonical execution is called a *canonical schedule* of an application.

For AND-model applications, all tasks will be executed during any execution and there is only *one* canonical execution, which corresponds to *one* canonical schedule. For example, suppose that the application in Figure 3.1a has a single deadline at time 20, under ER-LTF priority assignment heuristic, the application's canonical schedule on two processing units is shown in Figure 4.1a.

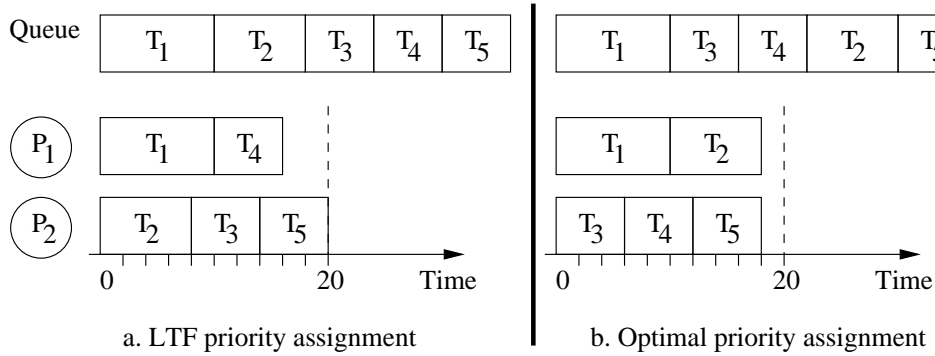


Figure 4.1: Canonical schedules of an AND-Model application

In the figure, the order of tasks in the queue follows their priorities. For the schedules discussed in this work, the height of the rectangles represents processor frequency and the X-axis represents time. The area of a rectangle represents the worst case computation requirement (e.g., the number of cycles) needed to execute the corresponding task. Notice that the optimal priority assignment could result in a shorter schedule as shown in Figure 4.1b.

For AND/OR-model applications, only a subset of tasks will be invoked in any specific execution. An *execution path* is defined as a *distinguished* subset of tasks that are invoked in one execution and there are generally more than one execution path in an AND/OR-model application. Notice that *any* task in an application will be in *at least* one execution trace.

An *integrated segment* of an application is defined as the tasks that are between two adjacent OR nodes and in the same execution path. The tasks in an integrated segment, as a whole, will be executed or will not be executed. Notice that, there may be more than one integrated segments between two adjacent OR nodes. The concept of canonical schedule can be applied to integrated segments in an AND/OR application and a *canonical schedule section* can be obtained for each integrated segment under a given priority assignment heuristic. In what follows, we use canonical schedule to refer to all canonical schedule sections along the longest path in an AND/OR-model application.

For the AND/OR-model application shown in Figure 3.4, the canonical schedule on two processing units under ER-LTF priority assignment heuristic is illustrated in Figure 4.2. Note that the synchronization nodes are considered as dummy tasks and are shown as bold vertical lines. The dotted rectangles represent the integrated segments.

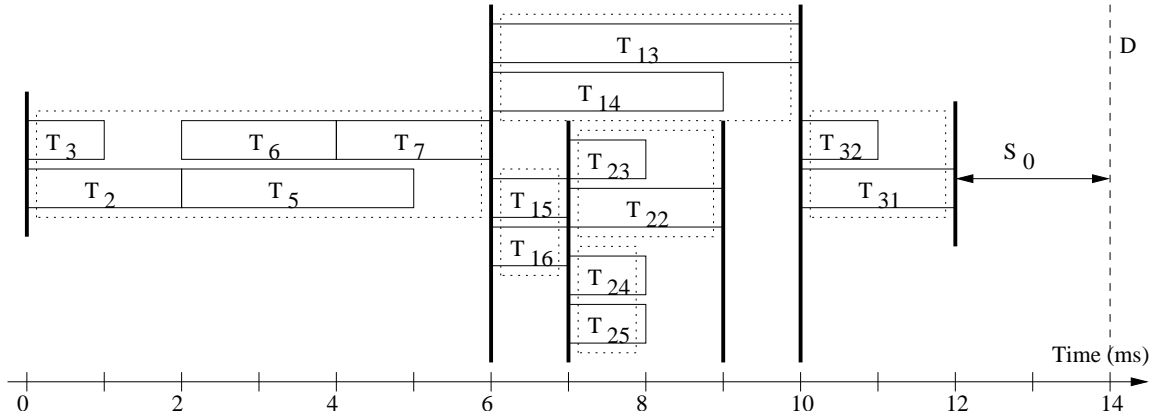


Figure 4.2: Canonical schedule of an AND/OR-Model application

4.1.3 Importance of Execution Order

Intuitively, if all tasks use no more than their WCET during an execution instance, an application should finish earlier than the canonical schedule in which all tasks use their WCET. However, for dependent tasks, due to priority inversion caused by tasks' dynamic behavior, an application may use more time than its canonical schedule, as will be demonstrated in Section 4.1.3.2.

4.1.3.1 Independent Tasks Recall that we define the priority of a task T_i as a tuple of its ready time rt_i and WCET c_i . For independent tasks, all of them are ready at the same time (e.g., the beginning of a frame) and their priorities are solely determined by their WCET under ER-LTF priority assignment heuristic. Thus, the order of tasks being dispatched and executed (i.e., the order of tasks in the global queue) does not depend on tasks' run-time behavior and are the same for all executions. However, due to the run-time behavior of tasks, they may be dispatched to and executed on processing units that are different than the ones in the canonical schedule.

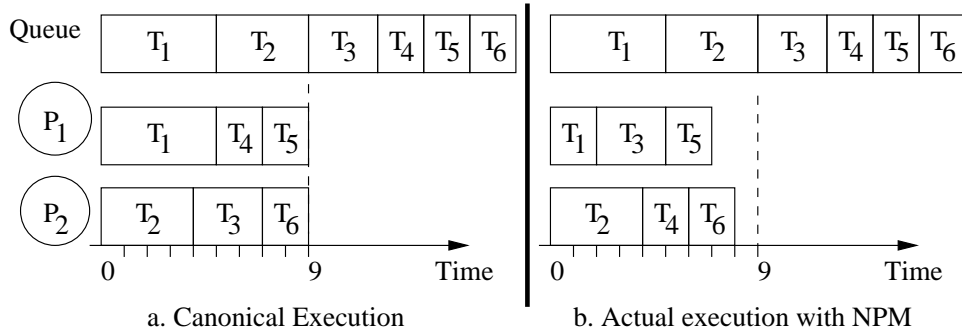


Figure 4.3: The canonical schedule and one running of an independent task set

Consider an independent task set that has six tasks: $\Gamma = \{T_1, T_2, T_3, T_4, T_5, T_6\}$ and a common deadline $D = 9$, where $T_1 = (5, 2)$, $T_2 = (4, 4)$, $T_3 = (3, 3)$, $T_4 = (2, 2)$, $T_5 = (2, 2)$, $T_6 = (2, 2)$. When they are executed on a shared memory system that has two processing units, Figures 4.3a and 4.3b show the canonical schedule and one actual execution where tasks take their average case execution time (ACET), respectively. From the figures, we can see that the order of tasks in the global queues are the same (i.e., tasks have the same priorities) in both cases. Moreover, both cases can finish executing all tasks before the deadline.

However, for the case where tasks take their ACET, T_1 finishes earlier at time 2 and the first processing unit fetches from the global queue the next highest priority task, which is T_3 . Without reclaiming the slack, T_3 will finish its execution at time 5. It is different from the canonical schedule, where T_3 runs on the second processing unit from time 4 to time 7. Since both start and end time of task T_3 become earlier, it in turn leads to earlier start

and completion of the remaining tasks. Thus, the actual running takes less time than the canonical schedule.

In what follows, we prove that, for independent tasks that have *fixed* priorities (i.e., the order of tasks being dispatched and executed is fixed), if the timing constraints can be met in the canonical schedule, any execution of an application where tasks take no more time than their WCET can meet the timing constraints.

First, we define some notations. In addition to the *ready time* rt_i of task T_i , st_i and ft_i are defined as T_i 's *start time* and *finish time* during one execution of an application, respectively. For the canonical execution, rt_i^c , st_i^c and ft_i^c are correspondingly defined as the *canonical ready time*, *canonical start time* and *canonical finish time* of T_i . The *actual execution time* of T_i during an execution is denoted by aet_i . Furthermore, the *estimated finish time* of task T_i is defined as $eft_i = st_i + c_i$. We denote ect_k by the *estimated completion time* of the k^{th} processing unit, which is the estimated finish time of the task that is currently running on that processing unit.

From the assumption of tasks' WCET, if all tasks can begin their execution no later than their canonical start time during an execution instance, they will finish their execution no later than their canonical finish time. This means that the execution can meet an application's timing constraints if the canonical execution could.

Lemma 1. *For an application consisting of a set of independent tasks $\Gamma = \{T_1, \dots, T_n\}$, if the priorities of tasks are fixed, the start time of tasks during an execution instance in which tasks use no more than their WCET will be no later than their canonical start time. That is, $st_i \leq st_i^c$ ($i = 1, \dots, n$).*

Proof. Suppose there are M (≥ 1) processing units. If the number of tasks is no more than the number of processing units (i.e., $n \leq M$), all tasks will start their execution at time 0 during any execution instance, that is $st_i = st_i^c = 0$ ($i = 1, \dots, n$). The result is trivial.

Next, we consider the case where $n > M$. The proof is by induction on $T_i, i = 1, \dots, n$. Without loss of generality, we assume that the priority of task T_i is higher than that of T_{i+1} ($i = 1, \dots, n - 1$). That is, tasks with lower identification numbers have higher priorities and are closer to the front of the global ready queue.

Base case: When $n > M$, initially, each of the M processing units fetches one task for execution. Thus, for the first M tasks, there is $st_i = st_i^c = 0$ ($i = 1, \dots, M$).

Induction step: Assume that T_{j-1} ($j - 1 \geq M$) is the most recently started task and $st_i \leq st_i^c$ for $i = 1, \dots, j - 1$. The next highest priority task (i.e., the header task in the global ready queue) to be fetched and executed will be T_j .

From the definition, for the first $j - 1$ tasks, we have:

$$\begin{aligned} ft_i &= st_i + aet_i \\ eft_i &= st_i + c_i \\ ft_i^c &= st_i^c + c_i \end{aligned}$$

When tasks use no more time than their WCET, that is, $aet_i \leq c_i$, we will have:

$$ft_i \leq eft_i \leq ft_i^c; i = 1, \dots, j - 1$$

From the algorithm and definitions, after T_{j-1} starts and before any other task finishes, the estimated completion time of one processing unit equals the estimated finish time of the active task that currently runs on that processing unit. That is,

$$ect_k = eft_{k_i}$$

where $k = 1, \dots, M$ and $k_i = 1, \dots, j - 1$. T_{k_i} is the task currently running on the k^{th} processing unit. Define the *active task set* $\hat{\Gamma}$ to be the set of tasks currently running on all processing units (i.e., $\hat{\Gamma} = \{T_{k_i}\}$). Moreover $\max_M\{X_1, \dots, X_n\}$ ($M \leq n$) is defined as the set that contains the M largest elements in $\{X_1, \dots, X_n\}$.

Without loss of generality, suppose task T_{l_i} on the l^{th} processing unit ($l = 1, \dots, M$ and $l_i = 1, \dots, j - 1$) is the next task that finishes its execution. Under global scheduling, the l^{th} processing unit will fetch and execute task T_j . From the algorithm and the fact that tasks use no more time than their WCET, we have:

$$st_j \leq \min\{eft_i | T_i \in \hat{\Gamma}\}$$

Notice that,

$$\min\{eft_i | T_i \in \widehat{\Gamma}\} \leq \min\{\max_M\{eft_i | i = 1, \dots, j-1\}\}$$

Thus,

$$st_j \leq \min\{\max_M\{eft_i | i = 1, \dots, j-1\}\}$$

For the case of canonical execution, we have:

$$st_j^c = \min\{\max_M\{ft_i^c | i = 1, \dots, j-1\}\}$$

From above discussion, $eft_i \leq ft_i^c$ ($i = 1, \dots, j-1$). Thus,

$$\min\{\max_M\{eft_i | i = 1, \dots, j-1\}\} \leq \min\{\max_M\{ft_i^c | i = 1, \dots, j-1\}\}$$

Therefore, $st_j \leq st_j^c$. That is, the start time of task T_j is no later than its canonical start time.

In summary, if independent tasks in an application have the same priorities as in canonical execution and use no more than their WCET, they will start their execution no later than their canonical start time during any execution instance.

□

From Lemma 1, it is trivial to prove the following theorem.

Theorem 1. *For an application consisting of independent tasks that have the same priorities during any execution, if the timing constraints can be met in the application's canonical execution, any execution of the application can meet the timing constraints.*

For independent tasks, there is no precedence constraint and all tasks are ready at the very beginning of a frame. That is, $rt_i^c = rt_i = 0$ ($i = 1, \dots, n$). Thus, the priorities of tasks under ER-LTF heuristic only depend on their WCET and are the same for all execution instances. Therefore, under ER-LTF heuristic, an application that consists of independent tasks finishes its execution no later than the case of canonical execution if tasks use no more than their WCET.

4.1.3.2 Anomaly of List Scheduling for Dependent Tasks For dependent tasks, recall that list scheduling puts a task into the global queue *whenever* all its predecessors finish their execution. Due to the run-time behavior of tasks, the ready time of a task may be different in different execution instances, which results in different priorities and different orders of tasks being added to the global queue. Therefore, an application that consists of dependent tasks may *not* be able to meet the timing constraints even if the application's canonical schedule does.

Consider the application with precedence constraints as shown in Figure 3.1b, when it is executed on two processing units, its canonical schedule is shown in Figure 4.4a. When all tasks use their WCET, we can see that T_1 and T_2 are root tasks and ready at time 0. T_3 and T_4 are ready at time 2 when their predecessor T_1 finishes execution. T_5 is ready at time 3 and T_6 is ready at time 6. Suppose that the application has a deadline at time 12.

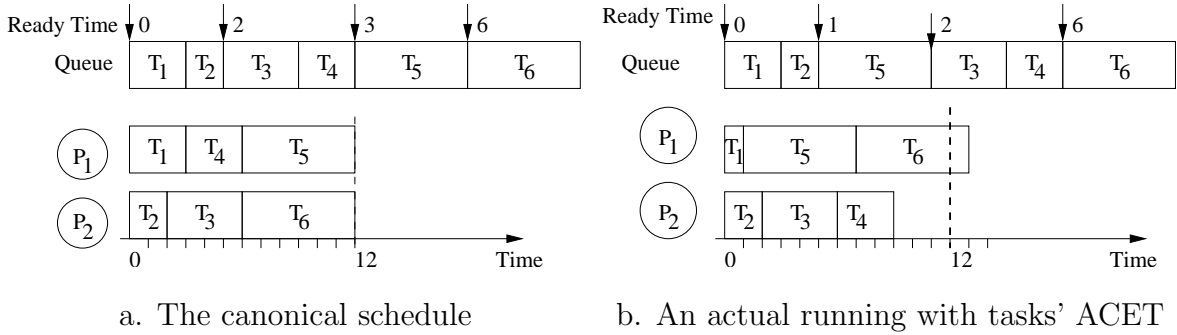


Figure 4.4: The canonical schedule and one running of one dependent task set

If every task uses its average case execution time (ACET), Figure 4.4b shows the execution instance of the application. From the figure, it is clear that the execution does not finish by the application's deadline. This comes from the fact that T_5 's priority is different from the one in the canonical schedule due to the earlier completion of T_1 . When T_5 is put into the global queue and is executed before T_3 and T_4 , which is different from the canonical execution, it causes the mapping and scheduling of T_3 , T_4 and T_6 to be different from the canonical schedule and in turn leads to the late completion of tasks and a deadline miss.

4.1.3.3 List Scheduling with Fixed Priority Notice that the priorities of independent tasks do not depend on the tasks' run-time behavior and, the scheduling for independent tasks does not exhibit the anomaly of using more time than their canonical schedule. Based on this observation, we will use *fixed-priority* list scheduling to prevent scheduling of dependent tasks from the run-time uncertainties and anomalies. That is, during run-time, the priorities of tasks are kept the same as the ones in the canonical schedule.

Define the *canonical priority* of a dependent task T_i as a tuple (rt_i^c, c_i) , where rt_i^c is the *canonical ready time* of T_i in the canonical schedule and c_i is T_i 's WCET. Since the canonical schedule of an application can be determined statically, the canonical priorities of tasks are fixed and do not dependent on tasks' run-time behaviors. Thus, instead of waiting until tasks are ready before putting them into the global queue, we can add all tasks into the global queue in the order of their canonical priorities at the very beginning of a frame. Whenever a processing unit is free, it will check the readiness of the task at the head of the queue, which has the highest priority among all remaining tasks. If the task at the head of the queue is ready, the free processing unit will fetch and execute it; otherwise, the free processing unit will go to idle although ready tasks with lower priorities may exist.

For the example in Figure 4.4, with each task using its ACET and canonical priority, the execution is illustrated in Figure 4.5. By keeping the same priorities as in canonical schedule, the execution meets the application's deadline. Notice that the first processing unit is idle during the second time unit since the header task T_3 is not ready when the processing unit is free at time 1.

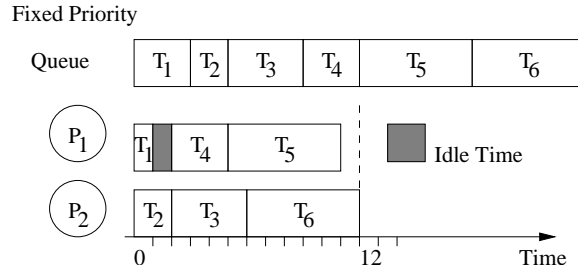


Figure 4.5: Fixed-priority list scheduling for dependent tasks

For an application that consists of a set of dependent tasks, if the tasks are executed

according to their canonical priorities and use no more time than their WCET, we will prove that any execution under list scheduling can meet the timing constraints if these timing constraints can be met in the canonical schedule.

Lemma 2. *Consider an application that consists of a set of dependent tasks. For any task T_i during any execution under fixed-priority list scheduling with each task keeping its canonical priority and using no more time than its WCET, there is a point in time t_i that is no later than T_i 's canonical start time st_i^c , and at which the following two conditions are satisfied: (a) T_i is the first task in the global queue and is ready (i.e., all its predecessors finished execution); and (b) a free processor is available to fetch and execute T_i . That is, $st_i = t_i \leq st_i^c$.*

Proof. Suppose that the application consists of n tasks, that is, $\Gamma = \{T_1, \dots, T_n\}$. Without loss of generality, we assume that the canonical priority of task T_i is greater than that of task T_{i+1} ($i = 1, \dots, n-1$). That is, the order of tasks in the global queue follows the inverse order of tasks' identification number. Tasks with smaller identification are closer to the front of the global queue and thus are dispatched and executed earlier.

The proof then proceeds by induction on i , for task T_i in the application.

Base case: Suppose that h ($\leq M$) root tasks of the application begin execution at time 0, where M is the number of processing units in the system. Hence, $st_i = st_i^c = 0$ for $i = 1, \dots, h$.

If $h < M$, this means that there are only h root tasks and $M - h$ processing units will be idle at time 0; otherwise, if $h = M$, the number of root tasks is at least M , the number of processing units.

Induction step: Assuming that T_{j-1} is the most recently started task and $st_i \leq st_i^c$ for $i = 1, \dots, j-1$.

(a) From the scheduling algorithm and tasks' priorities, after the first $j-1$ tasks are dispatched, task T_j becomes the header task in the global queue. For any predecessor, T_q , of task T_j (i.e., $T_q \rightarrow T_j \in E$), we have $1 \leq q \leq j-1$.

Hence, in the canonical schedule, task T_q finishes no later than st_j^c , that is, $ft_q^c \leq st_j^c$. During actual runnings, task T_q starts execution at $st_q \leq st_q^c$ and finishes no later than $ft_q = st_q + c_q \leq st_q^c + c_q = ft_q^c \leq st_j^c$. Therefore, task T_j is ready no later than st_j^c .

(b) Next, for the canonical schedule, before task T_j starts at time st_j^c , there are at most $x = M - 1$ tasks (among the first $j - 1$ tasks) that are running and will finish later than st_j^c (since at least one processing unit is free and fetches T_j at time st_j^c). For task T_a ($1 \leq a \leq j - 1$) that finishes no later than st_j^c in the canonical schedule, we have $ft_a^c \leq st_j^c$.

During actual runnings, we have $f_a = st_a + c_a \leq st_a^c + c_a = ft_a^c$, that is, T_a will finish no later than $ft_a^c \leq st_j^c$. Thus, there are at most $x = M - 1$ tasks that could finish later than st_j^c . That is, at or before time st_j^c , at least $M - x = 1$ processors are idle and free.

Thus, one free processor will fetch and execute task T_j no later than st_j^c . That is, $st_j \leq st_j^c$.

Therefore, under fixed-priority list scheduling with tasks taking their canonical priorities and use no more time than their WCET, for any task T_i in an application, there is $st_i \leq st_i^c$. \square

From Lemma 2, it is straightforward to prove the following theorem.

Theorem 2. *For an application that consists of a set of dependent tasks, if tasks use at most their WCET and the application's canonical execution can meet the timing constraints, any execution of the application under fixed-priority list scheduling with each task keeping its canonical priority can meet the application's timing constraints.*

4.2 STATIC POWER MANAGEMENT (SPM)

We will start discussing power management schemes for the AND-model applications and then extend them to the AND/OR-model applications. If the length of an application's canonical schedule is less than the application's deadline, we can take advantage of the *static slack*, which is defined as the difference between the deadline and the canonical schedule length. For example, there are 2 time units of static slack for the cases of both Figure 4.1b and Figure 4.2.

Assuming that tasks use their WCET, static energy management explores static slack to scale down the processing frequency and supply voltage to save energy. The slack can be allocated to one task or to all tasks [62, 88, 28] in different static energy management

schemes. For illustration purpose, we consider a simple example that only has three tasks as shown in Figure 4.6a. Suppose that the application is executed on two processing units and has a deadline at time 6. The canonical schedule is shown in Figure 4.6b, where the static slack is $S_0 = 2$.

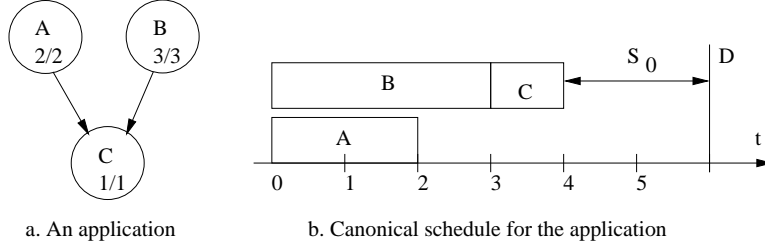


Figure 4.6: A simple example and its canonical schedule

4.2.1 Greedy Static Power Management (G-SPM)

G-SPM allocates all static slack to the first task running on each processing unit. It can be implemented in two steps. First, the schedule is shifted forward (that is, toward the deadline) by a time equal to the amount of slack. That is, the start and end times for tasks are all delayed by the same amount of time and the precedence and synchronization constraints are maintained. Then, the first task on each processing unit can claim the time between the beginning of a frame and its start time while obeying any timing constraints.

Applying G-SPM to the application in Figure 4.6a, both tasks A and B will get 2 units of slack and can scale down their processing frequencies and voltages proportionally. The static schedule is shown in Figure 4.7a where each task is labeled by its processing frequency.

4.2.2 Uniform Static Power Management (U-SPM)

As discussed in [39], because of the convex relation between processor dynamic power and processing frequency, the optimal schedule that results in the minimum energy consumption for uniprocessor systems is to run all tasks with the *same* processing frequency. That is,

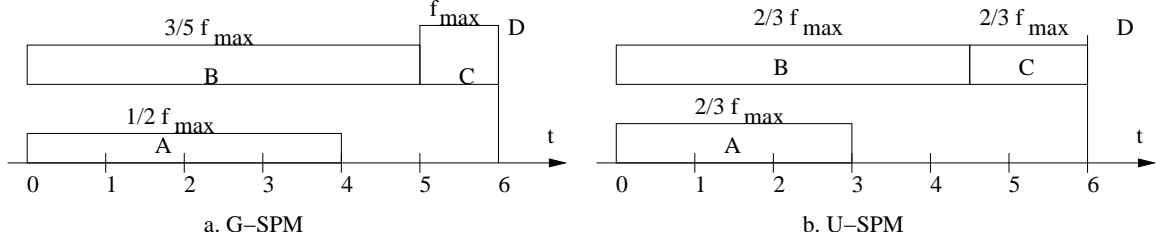


Figure 4.7: Two simple static energy management schemes

the optimal static slack allocation is to proportionally distribute static slack among all task according to their WCET. Following this idea, a *uniform static power management (U-SPM)* scheme for parallel systems is proposed in which static slack is distributed over a schedule evenly [28]. Therefore, tasks under the U-SPM scheme will run at the same frequency, which is the ratio of an application's canonical schedule length over its deadline.

For the above example, the schedule under the U-SPM scheme is shown in Figure 4.7b. Here, the canonical schedule length is 4 and there are 2 units of slack time. Thus, all tasks run at frequency $f = \frac{2}{3}f_{max}$. Recall that m is assumed to be 3. Therefore, the energy consumption is quadratically related to processing frequency (see Equation 3.4) and the energy consumed will be $\frac{4}{9}E$, where E is the energy consumed when no power management (NPM) is applied.

Although U-SPM is optimal for uni-processor systems in terms of static energy savings [39], it is not optimal for parallel systems. The reason is that, gaps may exist in a schedule due to dependence between tasks and not all processing units have tasks for execution all the time.

Notice that a schedule can be divided into many *schedule sections* separated by the canonical start and end times of the tasks. The *degree of parallelism (DP)* of a schedule section is defined as the number of processing units that have tasks for execution during that section. By uniformly stretching the whole schedule, U-SPM does not consider different DPs of different schedule sections. In fact, in the above example, U-SPM consumes even more energy than G-SPM, which consumes only $\frac{29}{150}E$ units of energy. This is because G-SPM happens to allocate more slack to schedule sections that have higher DPs (and thus more energy efficient as discussed in Section 4.2.3).

4.2.3 Static Power Management with Parallelism (SPM-P)

Intuitively, more energy savings can be obtained by giving more slack to schedule sections that have higher DPs because the same amount of slack could be used to scale down more computation. In this section, we propose a scheme named *static power management with parallelism (SPM-P)* which takes into consideration the different DPs of different sections in a schedule when allocating static slack.

4.2.3.1 SPM-P for Dual-Processor Systems Starting from the simplest parallel systems that have only two processing units, the DP in a schedule could range from 1 to 2. Notice that, excluding the slack time at the end of a schedule, there is no schedule section that has $DP = 0$ since inter-processor communication is assumed to be instantaneous for shared-memory systems.

For efficient allocation of static slack, an application's canonical schedule will first be partitioned into sections. For the schedule in Figure 4.6b, the first two time units have $DP = 2$, and the third and fourth time units have $DP = 1$. We define L_{ij} as the length of the j^{th} section that has $DP = i$. The total length of schedule sections that have $DP = i$ in a schedule is defined as $TL_i = \sum_j L_{ij}$. The schedule in Figure 4.6b for the simple application will be partitioned as shown in Figure 4.8. Here, we have $TL_1 = 2$ and $TL_2 = 2$.

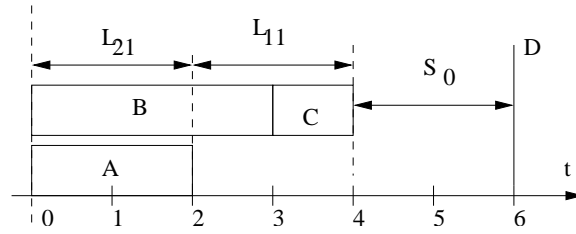


Figure 4.8: Parallelism in the schedule for a simple application

In general, suppose that an application running on a dual-processor system has static slack of S_0 . The total length of schedule sections with parallelism of i is TL_i ($i = 1, 2$). With s_i defined as the amount of static slack allocated to sections that have $DP = i$ ($i = 1, 2$),

the total energy consumption E after allocating the static slack S_0 will be:

$$\begin{aligned}
E &= \sum_i E_i = \sum_i (i \cdot C_{ef} \cdot f_i^3 \cdot (TL_i + s_i)) \\
&= \sum_i \left(i \cdot C_{ef} \left(\frac{TL_i}{TL_i + s_i} f_{max} \right)^3 (TL_i + s_i) \right) \\
&= C_{ef} \cdot f_{max}^3 \sum_i \left(i \cdot \frac{TL_i^3}{(TL_i + s_i)^2} \right)
\end{aligned} \tag{4.1}$$

where f_i is the frequency for executing schedule sections that have $DP = i$ ($i = 1, 2$). From last equation, we can see that E decreases monotonically when s_i increases. Therefore, all the slack of S_0 should be allocated. To efficiently allocate S_0 , we need to minimize E subject to:

$$0 \leq s_i \tag{4.2}$$

$$\sum_i s_i = S_0 \tag{4.3}$$

where $i = 1, 2$. The constraints put limitations on how to allocate the static slack S_0 . Setting

$\frac{\partial E}{\partial s_1} = \frac{\partial E}{\partial s_2} = 0$, we can get the following solutions:

$$s_1 = \frac{TL_1(S_0 - (2^{1/3} - 1)TL_2)}{TL_1 + 2^{1/3}TL_2}; \tag{4.4}$$

$$s_2 = \frac{TL_2(2^{1/3}S_0 + (2^{1/3} - 1)TL_2)}{TL_1 + 2^{1/3}TL_2}; \tag{4.5}$$

From the solutions, if $S_0 \leq (2^{1/3} - 1)TL_2$, we set $s_1 = 0$ since there is a constraint that $s_i \geq 0$. Thus, $s_2 = S_0$, that is, all the static slack will be allocated to the sections with parallelism of 2.

For the application in Figure 4.6a, we have $s_1 = 0.6550$ and $s_2 = 1.3450$. The energy consumption is computed as $0.3083E$. Compared with the energy consumption when using U-SPM $\frac{4}{9}E = 0.4444E$, an additional 30% energy is saved by SPM-P.

4.2.3.2 SPM-P for M-Processor Systems The above idea can be easily extended to M-processor systems. Assuming that there are M processors in a system, the degree of parallelism (DP) in a schedule will range from 1 to M . Suppose that schedule sections with $DP = i$ have total length of TL_i (which may consist of several sub-sections L_{ij} , $j = 1, \dots, u_i$, where u_i is the total number of sub-sections that have $DP = i$) and the static slack in the system is S_0 . The amount of slack allocated to schedule sections with $DP = i$ is defined as s_i . The total energy consumption E after allocating S_0 would be the same as shown in Equation (4.1) with $i = 1, \dots, M$.

The problem of finding an optimal allocation of S_0 to TL_i in terms of energy consumption will be to find s_1, \dots, s_M so as to

$$\text{minimize}(E)$$

subject to:

$$0 \leq s_i \tag{4.6}$$

$$\sum_i s_i = S_0 \tag{4.7}$$

where $i = 1, \dots, M$.

Solving the above problem is similar to solving the constrained optimization problem presented in [7]. We can also approximate the solution by dividing slack S_0 into small shares with length δS (i.e., there will be $\frac{S_0}{\delta S}$ slack shares) and allocate the shares to different schedule sections. For each allocation, one slack share δS is allocated to schedule sections with $DP = i$ such that energy reduction ΔE_i ($i = 1, \dots, M$) is maximized:

$$\begin{aligned} \Delta E_i &= E_i - E'_i \\ &= i \cdot C_{ef} \left(f_i^3 TL_i - \left(\frac{TL_i}{TL_i + \delta S} f_i \right)^3 (TL_i + \delta S) \right) \\ &= i \cdot C_{ef} f_i^3 \frac{TL_i \delta S (2TL_i + \delta S)}{(TL_i + \delta S)^2} \end{aligned} \tag{4.8}$$

where E_i and E'_i are the energy consumptions for sections with $DP = i$ before and after getting δS , respectively.

The precision of this approximation process can be controlled by varying the size of each slack share. In general, the smaller δS is, the more accurate the solution would be. However, there will be more allocation steps and consume more time. In Section 4.6, compared with G-SPM and U-SPM, we will evaluate the effectiveness of SPM-P on energy savings.

4.2.4 SPM for AND/OR Applications

The difference between AND/OR-model applications and AND-model applications lies in the number of execution paths they have. While there is only one execution path in AND-model applications, more than one execution path generally exist in AND/OR-model applications. In addition to static slack, the amount of slack expected from executing the *shorter-than-longest* paths can also be determined statically by examining the difference on the canonical schedule length of different segments between adjacent OR nodes.

For those slacks, the scheme discussed above could be applied to each schedule segment to reclaim the slack statically. However, for applications that have nested OR nodes, reclaiming such slack could be an iterative process and may be computationally expensive.

Dynamic slack is generated when tasks of a real-time application use less than their WCET or shorter-than-longest paths are taken during the application's actual execution. Considering the large variations in the run-time behavior of real-time tasks, which only consume a small fraction (e.g., 10% to 40%) of their WCET in many cases [23], excessive amount of dynamic slack is expected. Thus, dynamic slack reclamation is necessary and the slack from executing different paths can also be considered as dynamic slack and reclaimed on-line. The on-line reclamation of such slack is discussed in Section 4.3.3.

4.3 DYNAMIC POWER MANAGEMENT (DPM)

In this section, we address the problem of reclaiming dynamic slack and propose several energy management schemes that scale down the processing frequency of processing units for energy savings without extending an application's schedule length. That is, if a specific scheduling algorithm can generate a feasible schedule for an application, applying our en-

ergy management schemes upon the scheduling algorithm will still ensure that all timing constraints of the application are met.

4.3.1 Infeasibility of Simple Greedy Slack Reclamation (GSR)

In [62], Mossé et al. proposed one greedy slack reclamation for uni-processor systems that allocates any available slack to the next ready task to be executed. However, when we apply the same idea to parallel systems that have multiple processing units, that is, any available slack on one processing unit is given to the next task running on that unit, it turns out that an application may miss its deadline even if the deadline can be met by the application's canonical schedule.

For the independent task set discussed in Section 4.1.3.1, Figure 4.9a shows that, when tasks use their ACET, task T_1 finishes its execution at time 2 on the first processing unit with 3 time units of slack. With GSR scheme, this slack is given to the next task T_3 that runs on that processing unit. Thus, T_3 gets 6 units of time in total including its WCET and the processor frequency is scaled down to $\frac{3}{6}f_{max}$ accordingly. When T_3 uses up its allocated time, T_6 misses the deadline D as illustrated in Figure 4.9b.

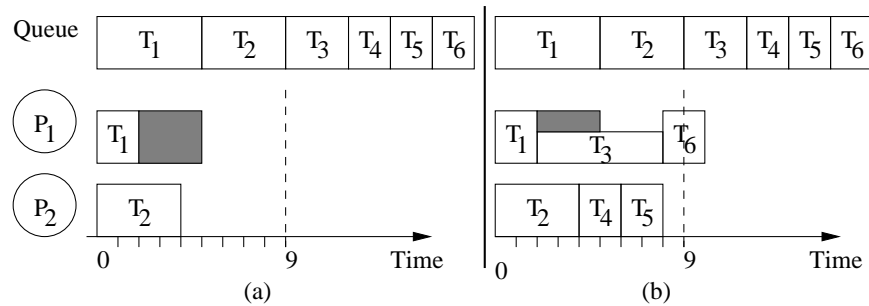


Figure 4.9: The simple greedy scheme

Therefore, even if an application's canonical schedule can meet the application's timing constraints, global scheduling with the simple greedy slack reclamation cannot guarantee that all timing constraints are met during actual executions. Note that this is true even if the tasks' priorities (as defined in Section 4.1) do not change. The reason for this anomaly is because the run-time mapping is different from the canonical mapping.

To ensure the timing constraints are met, in addition to canonical priorities of tasks, we can further fix the mapping of tasks to processing units as the one in an application’s canonical schedule. However, fixing the mapping is equivalent to partition scheduling with individual ready task queues, where each processing unit could reclaim the slack for energy savings. As we mentioned earlier in Section 4.1, expecting that global scheduling can automatically balance the actual workload among processing units in a system and result in more energy savings, we focus on global scheduling in this work. In what follows, we propose new energy management schemes that provide a set of safeguards to prevent this anomaly from happening without fixing the mapping.

4.3.2 Shared Slack Reclamation (SSR) for AND-model Applications

To prevent slack reclamation from violating the timing constraints of an application, we first propose a *shared slack reclamation (SSR)* scheme for AND-model applications. By sharing slack among processing units appropriately before reclaiming it, SSR guarantees to meet an application’s timing constraints while saving energy. Then, considering different execution paths in an AND/OR-model applications, an extended *shifted/shared slack reclamation (S/SSR)* scheme is proposed in Section 4.3.3.

There are two phases for the SSR scheme: an *offline phase* and an *on-line phase*. The offline phase is used to check the feasibility of an application’s canonical schedule under list scheduling with certain priority assignment heuristic and, if feasible, to obtain the canonical priorities for all tasks in the application. Recall that fixed-priority list scheduling is needed to guarantee that an application’s timing constraints are met during any execution (see Section 4.1.3). The on-line phase employs fixed-priority list scheduling and slack is shared and reclaimed appropriately for energy savings while meeting the application’s timing constraints. Before formally presenting the algorithm, we first illustrate how SSR works with two examples.

4.3.2.1 Two Examples Notice that, if the tasks in Figure 4.9 use their WCET, T_2 should finish its execution earlier than T_1 , and T_3 should follow T_2 and be executed on the

second processing unit as shown in the canonical schedule (Figure 4.3). However, when T_1 finishes its execution earlier, T_3 is dispatched to the first processing unit, which is different from the canonical schedule and leads to different mappings of remaining tasks to processing units as shown in Figure 4.3b. When T_3 reclaims all the 3 units of slack coming from the early completion of T_1 on the first processing unit following GSR scheme, it starts execution at time 2 at frequency $\frac{3}{6}f_{max}$ and finishes execution at time 8. After T_4 and T_5 start and finish their executions, there is only 1 time unit left, which causes T_6 to miss the deadline at time 9 (Figure 4.9b).

In this case, it would be better to *share* the 3 units of slack, which comes from T_1 's early completion, by splitting it into two parts: 2 units are given to T_3 on the first processing unit and 1 unit (which is *the difference between the finish time of T_1 and T_2 in the canonical schedule*) is shared with T_4 on the second processing unit. With *slack sharing*, T_3 starts at time 2, executes for 5 time units at the frequency of $\frac{3}{5}f_{max}$ and ends at time 7. T_4 starts at time 4, executes for 3 time units at the frequency of $\frac{2}{3}f_{max}$ and also ends at time 7. Thus, both T_5 and T_6 can meet the deadline.

Figure 4.10 demonstrates the operations of the SSR scheme. When the first processing unit finishes T_1 at time 2, it finds that it has 3 units of slack, but only 2 units of them are before the expected finish time of T_2 on the second processing unit based on T_2 's WCET. After fetching T_3 , the first processing unit gives 2 extra units (the amount of slack before T_2 's expected finish time) to T_3 and shares the remaining slack with the second processing unit.

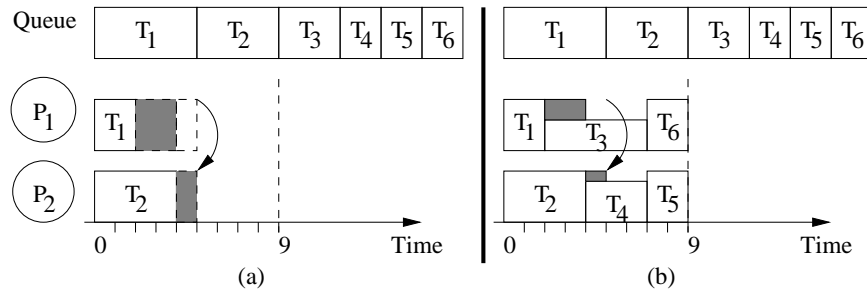


Figure 4.10: SSR for independent tasks

From a different point of view, sharing the slack may be looked at as T_1 being allocated

4 time units on the first processing unit instead of 5, while T_2 being allocated 5 time units on the second processing unit instead of 4. Thus, T_1 has 2 units of slack when it finishes early and T_2 will have 1 unit of slack when it finishes (since its WCET is 4). So, in some sense, the situation is similar to T_1 being assigned to the second processing unit and T_2 being assigned to the first processing unit, and all tasks that are assigned to the first processing unit in the canonical schedule will now be assigned to the second processing unit and *visa versa*.

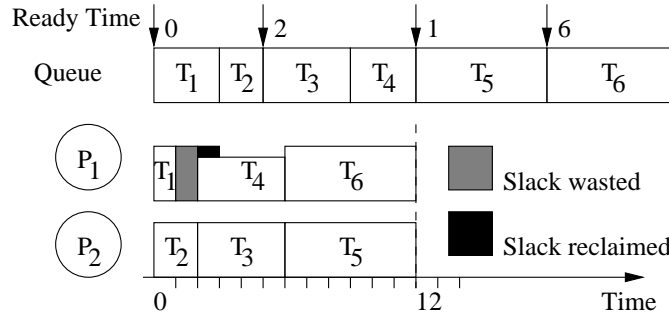


Figure 4.11: SSR for dependent tasks

For the case of dependent tasks, slack sharing under fixed-priority list scheduling for the application in Figure 3.1b is shown in Figure 4.11. As discussed in Section 4.1.3.3, in order to wait for the readiness of tasks T_3 and T_4 , the first processing unit becomes idle (or goes to sleep) during the second time unit and wastes part of its slack. However, by keeping the same priorities of tasks as in the application's canonical schedule, all tasks finish on time.

4.3.2.2 Offline Phase of SSR Scheme As discussed in Section 4.1.3, to ensure that an application's timing constraints are met, fixed-priority list scheduling is considered. For AND-model applications that consist of independent tasks, under ER-LTF heuristic, tasks' priorities can be determined by their WCET. However, to obtain tasks' priorities for applications that consist of dependent tasks, the emulation of the application's canonical execution is needed. Moreover, to collect the canonical timing information, it is also necessary to emulate an application's canonical execution.

In the offline phase of SSR scheme, the canonical execution of an application under list scheduling with ER-LTF heuristic on a specific system is emulated. Recall that, as one of

our assumptions, an application can meet its timing constraints in the canonical schedule. To facilitate the on-line phase of the SSR scheme, task T_i 's *canonical ready time* is collected (as rt_i^c) and is used to determine task T_i 's *canonical priority*. In addition, for each task T_i , the *canonical start time* and *canonical finish time* are recorded as st_i^c and ft_i^c , respectively.

Notice that, for frame-based applications, the offline phase of the SSR scheme is done only *once*. Then, the offline information can be used during the on-line phase of the SSR scheme for the execution of every frame.

4.3.2.3 On-line Phase of SSR Scheme After obtaining tasks' canonical priorities, at the beginning of SSR's on-line phase, all tasks are added into the global queue in the order of their priorities with higher priority tasks being closer to the front of the queue. To determine the *readiness* of tasks, we define the number of *unfinished immediate predecessors* for task T_i as uip_i . uip_i will decrease by 1 when any predecessor of task T_i finishes execution. Task T_i is ready when $uip_i = 0$. Recall that ect_k is the *estimated completion time* of the k^{th} processing unit.

Whenever a processing unit is free, it will check the header task of the global queue to see whether the task is ready or not. If the header task is ready, the processing unit will fetch and execute it; otherwise the processing unit becomes idle (or goes to sleep). The details of the algorithm are described below.

Algorithm 1 shows the on-line phase of the SSR scheme to process the tasks of an application in one frame. Each processing unit (PU_k) invokes the algorithm independently. Recall that the control information about tasks and processing units (e.g., the array of uip_i and ect_k) are kept in the shared memory and must be updated within a critical section. For simplicity, the mutual exclusive access to the critical section is not shown in the algorithm.

The algorithm is invoked on the k^{th} processing unit when a task finishes execution on PU_k , or when PU_k is sleeping and signaled by another processing unit. We use the function $wait()$ to put an idle processing unit to sleep and another function $signal(PU)$ to wake up a processing unit PU .

Initially, all tasks are put into *Global-Q* following the order of their canonical priorities

Algorithm 1 SSR on-line algorithm invoked by the k^{th} processing unit PU_k

```
1: Add all tasks into Global-Q with the priority of task  $T_i$  being  $(rt_i^c, c_i)$  ;
2: while (Global-Q is not empty) do
3:   if ( Head(Global-Q) is ready ) then
4:      $T_i = \text{Dequeue}(\text{Global-Q})$ ;
5:     Find a processing unit  $PU_r$  such that:
        $ect_r = \min\{ect_1, \dots, ect_N\}$ ;
6:     if ( $ect_k > ect_r$ ) then
7:        $ect_k \leftrightarrow ect_r$ ;
8:     end if
9:      $ect_k = st_i^c + c_i = ft_i^c$ ; /*as proved below  $t \leq st_i^{c*}$  /
10:     $f_i^g = \frac{c_i}{ft_i^c - t} f_{max}$  ;
11:    if ( (Head(Global-Q) is ready) AND ( $PU_s$  is sleep) ) then
12:       $\text{Signal}(PU_s)$ ;
13:    end if
14:    Execute  $T_i$  at frequency  $f_i^g$ ;
15:    for ( Each  $T_j$  such that  $T_i \rightarrow T_j \in E$  ) do
16:       $wip_j = wip_j - 1$ ;
17:    end for
18:  else
19:     $\text{wait}()$ ;
20:  end if
21: end while
```

(line 1). Recall that, under ER-LTF heuristic, the canonical priority of a task is defined as a tuple of the task's canonical ready time and its WCET. As discussed earlier in Section 4.1, it is important to have the fixed priority to ensure that the timing constraints of an application are met. The values of wip_i ($i = 1, \dots, n$) are set to the number of predecessors of task T_i and ect_k ($k = 1, \dots, N$) are set to 0 (not shown in the algorithm).

If the algorithm is invoked by a signal from another processing unit, it will begin at the 'waiting for signal' point (line 19). If the algorithm is invoked at the very beginning or when PU_k finishes executing a task, it starts from line 3. If the header task T_i of *Global-Q* is ready, PU_k fetches task T_i from *Global-Q* (line 4). Since $t \leq st_i^c$ (as proved later), PU_k reclaims the slack of $st_i^c - t$ (line 9) and calculates the frequency f_i^g to execute task T_i . If the new header task of *Global-Q* is ready, it signals one sleeping processing unit PU_s if any (line 11 and 12). Finally, PU_k executes T_i at the frequency of f_i^g (line 15).

4.3.2.4 Analysis of SSR Algorithm In Section 4.1, we have proved that, under fixed-priority list scheduling, an application can meet its timing constraints as in its canonical

schedule when no slack is reclaimed. Following the same approach, we prove in this section that SSR algorithm that reclaims slack and scales down the processing frequency of processing units for energy savings can also meet the timing constraints as in an application's canonical schedule.

From the above examples, we can see that the start time st_i of task T_i is no later than its canonical start time st_i^c with available slack $st_i^c - st_i \geq 0$. Thus, the frequency to execute task T_i is $f_i^g = \frac{c_i}{st_i^c - st_i + c_i} f_{max} \leq f_{max}$ (line 11 in Algorithm 1). Since f_i^g is the frequency that guarantees task T_i finishes no later than its canonical finish time ft_i^c if the computation requirement of T_i is no more than its worst case computation requirement, the application will meet its timing constraints if the application's canonical schedule does. Thus, to prove the correctness of Algorithm 1, we first show that any task T_i starts its execution no later than its canonical start time.

Lemma 3. *Under Algorithm 1, the start time st_i for any task T_i in an application is no later than its canonical start time. That is, $st_i \leq st_i^c$.*

Proof. The proof is similar to the one for Lemma 2.

Suppose that there are n tasks in the application considered. We will prove the lemma by shown that, for any T_i in the application, there is a time point that is no later than st_i^c and at which the following two conditions are satisfied: (a) T_i is header task of the global queue and ready (i.e., all its predecessors finished execution); and (b) a free processor is available for T_i .

Without loss of generality, we assume that the canonical priority of task T_i is greater than that of task T_{i+1} ($i = 1, \dots, n-1$). That is, the order of tasks in the global queue follows the inverse order of tasks' identification number. Tasks with smaller identifications are closer to the front of the global queue and are dispatched and executed earlier.

The proof then proceeds by induction on i , for task T_i in the application.

Base case: Suppose that h ($\leq M$) root tasks of the application begin execution at time 0, where M is the number of processing units in the system. Hence, $st_i = st_i^c = 0$ for $i = 1, \dots, h$.

If $h < M$, this means that there are only h root tasks and $M - h$ processing units will

be idle at time 0; otherwise, if $h = M$, the number of root tasks is at least M , the number of processing units.

Induction step: Assuming that T_{j-1} is the most recently started task and $st_i \leq st_i^c$ for $i = 1, \dots, j-1$.

(a) From the algorithm, after the first $j-1$ tasks are dispatched, task T_j becomes the header task of the global queue. For any predecessor, T_q , of task T_j (i.e., $T_q \rightarrow T_j \in E$), we have $1 \leq q \leq j-1$. Hence, in the canonical schedule, task T_q finishes no later than st_j^c , that is, $ft_q^c \leq st_j^c$.

During the on-line phase, task T_q starts execution at $st_q \leq st_q^c$ with frequency $f_q^g = \frac{c_q}{st_q^c - st_q + c_q} f_{max} \leq f_{max}$. If the computation requirement of T_q is no more than its worst case computation requirement, T_q will finish no later than $st_q + \frac{c_q}{f_q^g} = ft_q^c \leq st_j^c$. Therefore, task T_j is ready no later than st_j^c .

(b) Next, for the canonical schedule, before task T_j starts at time st_j^c , there are at most $x = M-1$ tasks (among the first $j-1$ tasks) that are running and will finish later than st_j^c (since at least one processing unit is free and fetches T_j at time st_j^c). For task T_a ($1 \leq a \leq j-1$) that finishes no later than st_j^c in the canonical schedule, we have $ft_a^c \leq st_j^c$. At run time, we have $f_a^g = \frac{c_a}{st_a^c - st_a + c_a} f_{max} = \frac{c_a}{ft_a^c - st_a} f_{max} \leq f_{max}$. That is, if the computation requirement of task T_q is no more than its worst case computation requirement, T_a will finish no later than $st_a + \frac{c_a}{f_a^g} = ft_a^c \leq st_j^c$. Thus, there are at most $x = M-1$ tasks that could finish later than st_j^c . That is, at or before time st_j^c , at least $M-x = 1$ processors are idle and free.

Thus, one free processor will fetch and execute task T_j no later than st_j^c and $st_j \leq st_j^c$.

Therefore, under Algorithm 1, for any task T_i in an application, there is $st_i \leq st_i^c$.

□

From Lemma 3, it is easy to get that, under the SSR algorithm, $ft_i \leq ft_i^c$ for any task T_i in an application. That is, the SSR algorithm can meet the timing constraints of an application as in the application's canonical schedule. Therefore, for the SSR algorithm, we can have the following theorem.

Theorem 3. *For an application with fixed task priorities, if the application's canonical sched-*

ule can meet the timing constraints, so can any execution under the SSR algorithm.

4.3.3 Shifted/Shared Slack Reclamation (S/SSR)

In the following, we consider energy management that further explores application's dynamic characteristics at the *task set level* (*shorter-than-longest execution path*) in addition to the *task level* (*less-than-maximum execution*). As discussed in Section 4.2, the slack from executing shorter paths can be determined statically and reclaimed by static energy management schemes. However, the static reclamation could be computation expensive. For applications exhibiting large variations on tasks' run-time behaviors, dynamic energy management is necessary for better energy savings. Thus, on-line reclamation of the slack from different paths is preferred.

In this section, we propose the *shifted/shared slack* reclamation that incorporates the characteristics of AND/OR-model applications and show how it is correct with respect to meeting an application's timing constraints. As for the SSR scheme, there are two phases: an *offline phase* and an *on-line phase*.

4.3.3.1 Offline Phase of S/SSR Scheme As for AND-model applications, the offline phase of the S/SSR scheme is used to collect the timing and priority information about the canonical execution of an AND/OR-model application that runs on a specific system. Again, the offline phase is done only once and the offline information will be used during the on-line phase of the S/SSR scheme for processing every frame of an application.

Notice that, compared with the AND-model where all tasks will be invoked during any execution, the AND/OR-model has more than one execution path and tasks to be invoked depend on which path is taken during an instance of execution. Thus, we cannot put all tasks of an AND/OR-model application into the global queue at the beginning of its execution.

In order to determine the amount of slack from different paths, the maximum schedule length for an application is computed as Π_c , which is the canonical schedule length along the longest path of the application. For each branch b_i after an OR node, the maximum remaining schedule length is computed as Π_c^i , which is the canonical schedule length along

the longest remaining path after the OR node following branch b_i . Moreover, to facilitate the discussion of the speculation schemes presented in Section 4.4, Π_a and Π_a^i are computed analogously as the *weighted average schedule length*, which assumes that tasks use their average case execution time and takes the probabilities of executing different paths into consideration.

For each task T_i , in addition to the canonical ready time rt_i^c , canonical start time st_i^c and canonical finish time ft_i^c , its *canonical execution order* is further recorded as eo_i^c . Notice that, the execution order of a task is determined by its priority. To ensure that an application's timing constraints are met, the priorities of tasks (i.e., the orders of tasks being executed) are kept the same as those in the canonical execution of the application. The execution order of an OR node is the maximum execution order of its predecessors plus 1. Tasks on different branches after an OR node will have the same execution order since they will not be executed during the same execution instance.

The offline phase is a two-pass process. To illustrate the concept and show how the algorithm works, we consider the AND/OR application shown in Figure 3.4. Recall that the synchronization nodes are considered as dummy tasks with zero execution time. In the first pass, using list scheduling with ER-LTF heuristic, the canonical schedules for all segments are generated as shown in Figure 4.2. The longest path of the application is shown bold in Figure 3.4 and, from the canonical schedule, $\Pi_c = 12ms$. Assuming that the application has a deadline $D = 14ms$, the initial slack is $S_0 = D - \Pi_c = 14 - 12 = 2ms$.

The canonical execution information is shown in Table 4.1. Notice that, the execution order of T_{29} (an OR node) is larger than the maximum execution order of its predecessors, T_{17} (with $eo_{17}^c = 14$) and T_{28} (with $eo_{28}^c = 20$), with $eo_{29}^c = \max\{eo_{17}^c, eo_{28}^c\} + 1 = \max\{14, 20\} + 1 = 21$. T_{11} and T_{12} are on different branches after T_{10} (an OR synchronization node) and will not be executed at the same time; they have the same execution order. The values of st_i^c are recorded as the time at which T_i starts execution.

The second pass of the offline phase prepares to steal the slack by *shifting* the canonical schedule as late as possible toward the application's deadline. For each task T_i , define the *shifted canonical start time* (sst_i^c) as the time at which T_i starts execution in the shifted canonical schedule. It is the latest time T_i can start execution to meet the application's

Table 4.1: Offline variables of an AND/OR-model application

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}
eo_i^c	1	2	3	4	5	6	7	8	9	10	11	11	12	13	12	13	14
st_i^c	0	0	0	2	2	2	4	6	6	6	6	6	6	6	6	6	10
sst_i^c	2	2	2	4	4	4	6	8	8	8	8	9	8	8	9	9	12
	T_{18}	T_{19}	T_{20}	T_{21}	T_{22}	T_{23}	T_{24}	T_{25}	T_{26}	T_{27}	T_{28}	T_{29}	T_{30}	T_{31}	T_{32}	T_{33}	
eo_i^c	14	15	16	16	17	18	17	18	19	19	20	21	22	23	24	25	
st_i^c	7	7	7	7	7	7	7	7	9	8	9	10	10	10	10	12	
sst_i^c	10	10	10	11	10	10	11	11	12	12	12	12	12	12	12	14	

timing constraints, provided that tasks in the same integrated segment have the same *shifting factor*, which is defined as the difference between a task's canonical start time and its shifted canonical start time. That is, if task T_j is in the same segment as T_i , $sst_j^c - st_j^c = sst_i^c - st_i^c$. Notice that the shifting is a recursive process when there are nested OR nodes. sst_i^c will be used to reclaim the slack for T_i at run time. The *shifted canonical finish time* for task T_i in the shifted canonical schedule is correspondingly recorded as $sft_i^c = sst_i^c + c_i$.

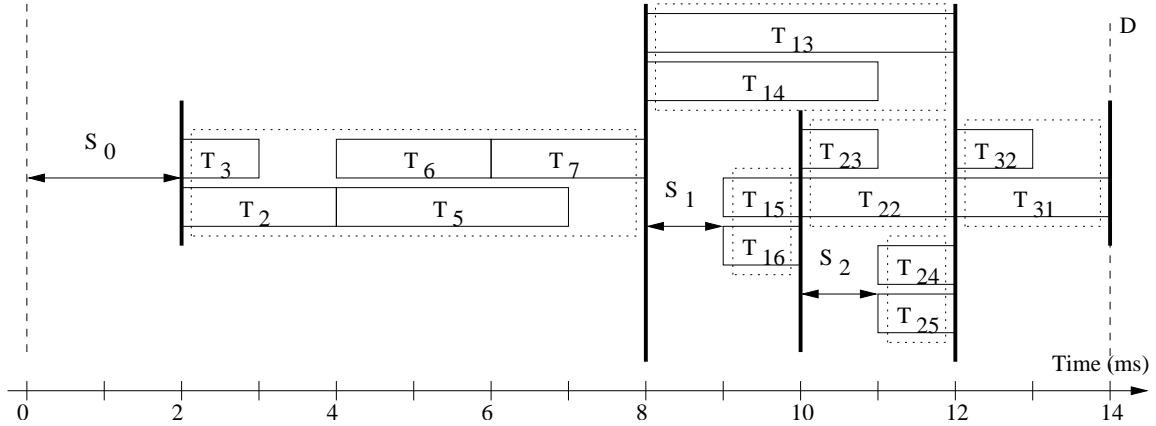


Figure 4.12: The shifted canonical schedules for an AND/OR application

The *shifted canonical schedule* for the example is shown in Figure 4.12, where S_0, S_1 and S_2 comprise the slack stolen by the algorithm. Notice that the tasks in one integrated segment are shifted together and have the same shifting factor. When there are nested OR nodes, different segments may have different shifting factors. We did not consider shifting tasks individually in a segment. For example, for the integrated segment consisting of task T_2, T_3, T_5, T_6 and T_7 , we may shift T_5 1ms more without violating the timing requirement.

But shifting single tasks increases the complexity of the offline algorithm and we do not expect too much gain from it since that $1ms$ could be reclaimed by the subsequent tasks in the on-line phase. After the schedule is shifted, sst_i^c is computed for every task T_i and the value of sst_i^c is also shown in Table 4.1.

4.3.3.2 On-line Phase of S/SSR Scheme To keep tasks having the same execution order as in an application's canonical schedule, the execution order of the next expected task (T_{NET}) is denoted by NET_EO . That is, $EO_{NET} = NET_EO$. The current time is represented by t . The frequency to execute task T_i under S/SSR is denoted as f_i^g .

Suppose task T_i starts execution at time t_i (i.e., the time at which one processor fetches T_i and starts to execute it). Recall that the *estimated finish time* (eft_i) for a task T_i is the time at which T_i is expected to finish its execution if it consumes all the time allocated to it. We have $eft_i = t_i + \frac{c_i}{f_i^g}$.

Initially, the tasks that have no predecessor (i.e., root tasks) are put into a *Ready-Q*. For any other task T_i , uip_i is initialized as the number of predecessors of T_i if the corresponding vertex is not an OR node, and 1 otherwise. The current time t is set to 0 and NET_EO is set to 1 (line 1).

The on-line algorithm of S/SSR scheme is shown in Algorithm 2. S/SSR is different from SSR, which is for AND-model applications, in the sense that S/SSR needs to handle the synchronization nodes and to add ready tasks to the global queue dynamically. For the successors of computation tasks and AND synchronization nodes, their $uips$ are updated properly and any successor task T_j will be put into *Ready-Q* when it is ready (i.e., $uip_j = 0$; from line 17 to 24). When entering *Ready-Q* (line 22), tasks are ordered in their canonical execution order eo_i^c . For an OR synchronization node, the first task in the chosen branch is put into *Ready-Q* (line 27 to 28). The same as SSR, the shared memory holds the control information, such as *Ready-Q* and the structure of uip , which must be updated within a critical section (not shown in the algorithm for simplicity).

For the AND/OR application in Figure 3.4, if an execution instance follows the lower branch at T_{10} and upper branch at T_{19} and the tasks on this path use their average case

Algorithm 2 S/SSR on-line algorithm invoked by PU_{id}

```
1: Add root tasks to Ready-Q and initialize  $uip_i$ ,  $t$  and  $NET\_EO$ ;
2: while (Ready-Q is not empty) do
3:    $T_i = \text{Head}(\text{Ready-Q})$ ;
4:   if ( ( $T_i$  is an OR node  $\parallel eo_k^c == NET\_EO$ )  $\&\&$  ( $uip_i == 0$ ) ) then
5:      $T_i = \text{dequeue}(\text{Ready-Q})$ ;
6:      $NET\_EO = NET\_EO + 1$  ;
7:     if ( $T_i$  is a computation node) then
8:       /* $T_i$  reclaims the slack  $sst_i^c - t$ ; as proved below,  $t \leq sst_i^c$ */
9:       /*total time allocated to  $T_i$  is  $sst_i^c - t + c_i = sft_i^c - t$ */
10:       $f_i^g = \frac{c_i}{(sft_i^c - t)}$ ; /*compute the frequency to execute  $T_i$ */
11:      /*Let  $T_{next} = \text{Head}(\text{Ready-Q})$ ; */
12:      if ( $PU_y$  is asleep  $\&\&$   $eo_{next}^c == NET\_EO$ ) then
13:         $\text{signal}(PU_y)$ ; /*Wake up one sleeping processor*/
14:      end if
15:      Execute  $T_i$  at frequency  $f_i^g$ ;
16:    end if
17:    if ( $T_i$  is a computation node or an AND node) then
18:      for ( each successor  $T_j$  of  $T_i$ ) do
19:         $uip_j = uip_j - 1$ ; /*update successor's variable*/
20:        if ( $uip_j == 0$ ) then
21:          /*put  $T_j$  into Ready-Q if it is ready*/
22:           $\text{enqueue}(T_j, \text{Ready-Q})$ ;
23:        end if
24:      end for
25:    else if ( $T_i$  is an OR node ) then
26:       $NET\_EO = eo_i^c + 1$ ; /*update the next expected task*/
27:      Let  $T_a$  be the first node in the path selected at the OR node
28:       $uip_a = 0$ ;
29:       $\text{enqueue}(T_a, \text{Ready-Q})$ ; /*put  $T_a$  into Ready-Q*/
30:    end if
31:  else
32:     $\text{wait}()$ ;
33:  end if
34:  /*go back to fetch a new task to execute*/
35: end while
```

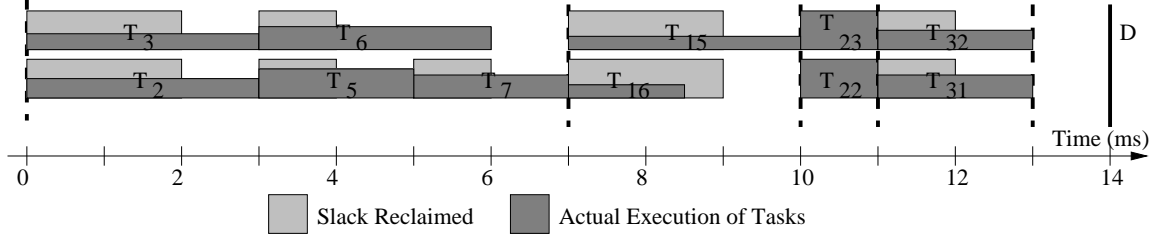


Figure 4.13: An actual execution of the AND/OR application

execution time, the single-instance execution trace is shown in Figure 4.13. Initially, both T_2 and T_3 reclaim $2ms$ of static slack (the difference between sst_i^c and the time they begin execution). From Algorithm 2 line 9, the frequency for T_2 and T_3 will be $\frac{f_{max}}{2}$ and $\frac{f_{max}}{3}$, respectively, and we have $eft_2 = 4ms$ and $eft_3 = 3ms$. Since T_2 only uses $\frac{3}{4}$ of the time allocated to it and T_3 uses all the time allocated to it, both T_2 and T_3 actually finish at time $3ms$. T_5 and T_6 get $1ms$ of slack each (again, the difference between sst_i^c and the time they begin execution), and are supposed to finish at time $7ms$ and $6ms$, at frequencies $\frac{3}{4}f_{max}$ and $\frac{2}{3}f_{max}$, respectively. T_7 is supposed to follow T_6 on the upper processor. Since T_5 only uses $\frac{1}{2}$ of the time allocated to it and T_6 uses all its time, T_5 actually finishes earlier than T_6 and T_7 follows T_5 on the lower processor and reclaims $1ms$ of slack. Finally, the application finishes its execution at time $13ms$, $1ms$ before its deadline.

4.3.3.3 Analysis of S/SSR Algorithm Recall that an execution path is defined as a set of tasks that are executed during an execution instance of an AND/OR application. Notice that the AND/OR synchronization nodes are dummy tasks with zero execution time. After properly removing the synchronization nodes in one path, the remaining computation tasks can be treated as a simplified AND-model application. For *any* execution path, the computation tasks under S/SSR will have the same timing as under SSR. From the proof of SSR as shown in Section 4.3.2, we can easily get the following theorem.

Theorem 4. *If the canonical schedules of an AND/OR application can meet the application's timing constraints, any execution of the application under Algorithm 2 will meet the timing constraints.*

While the shifted/shared slack reclamation scheme can guarantee to meet an application's timing constraints, there may be too many frequency changes since a new frequency is computed for each task. It is known that if all tasks execute at the same frequency, the minimum energy consumption can be achieved [39]. Considering frequency change overhead, the single frequency setting is even more attractive. From this intuition, using the statistical information about an application, we propose the following speculative algorithms.

4.4 SPECULATIVE SCHEMES

Based on different strategies, we developed two speculative schemes. One is to *statically predict* an optimal frequency. The second is to *dynamically adjust* the predicted optimal frequency while speculating about the remaining work. Since large amounts of slack can be expected from different branches after an OR synchronization node, the dynamic speculation is only performed after the OR synchronization nodes.

Although we can speculate before each task on the fly for uniprocessor systems [6], it is hard to implement it for parallel systems because of the difficulty of computing the remaining work in the system. The reason is that when a task ends on one processing unit, we do not know on which processing unit other tasks will run and some tasks may be in the middle of their execution.

4.4.1 A Static Speculation Scheme

For the static speculative scheme, the frequency at which an application should run, is decided at the very beginning of the application based on the statistical information about the whole application as:

$$f_{ss} = \frac{\Pi_a}{D} \quad (4.9)$$

where Π_a is the average schedule length of the application.

Even after f_{ss} is calculated, we choose the maximum frequency between f_{ss} and f_i^g for task T_i , where f_i^g is computed from the S/SSR on-line algorithm. This is to guarantee

temporal correctness, since the speculative frequency is optimistic and does not take tasks' worst case behaviors into consideration.

4.4.2 An Adaptive Speculative Scheme

If the statistical characteristics of tasks in an application vary substantially (e.g., the tasks at the beginning of one application have the average/maximal computation requirement ratio as 0.9 while tasks at the end of the application have the ratio as 0.1), it may be better to re-speculate the frequency while the application execution progresses based on the statistical information about the remaining tasks.

At OR nodes, processing units will synchronize and we can speculate a new optimal frequency for the remaining tasks at that moment. Using the statistical information about the remaining tasks on the executed path after an OR node, the speculative frequency would be set as:

$$f_{as} = \frac{\Pi_a^i}{D - t} \quad (4.10)$$

where t is the current time (the time when the OR node is processed) and Π_a^i is the average execution time needed when branch b_i is taken. Again, to guarantee the deadline, the frequency f_i for T_i will be: $f_i = \max(f_i^g, f_{as})$.

Because the speculative schemes never set a frequency below the one determined by S/SSR, they will meet the timing constraints if S/SSR can finish on time. Therefore, from the discussion in Section 4.3, the speculative schemes can meet an application's timing constraints if the application's canonical schedule under the same heuristic finishes on time.

4.5 PRACTICAL CONSIDERATIONS IN ENERGY MANAGEMENT

In the above discussion, we assumed that frequency¹ is continuous and ignored the overhead for frequency changes. However, current variable frequency processors have only a few frequency levels and frequency adjustment takes time and consumes energy [19, 34, 37].

¹Recall that supply voltage is reduced for lower frequencies under voltage scaling techniques and frequency changes are used to stand for changing both processing frequency and supply voltage.

Moreover, for global scheduling in shared memory systems, shared memory access contention should be considered. In what follows, we discuss how to incorporate these factors into our energy aware scheduling algorithms.

4.5.1 Overhead of Frequency Adjustment

There are two kinds of overhead that have to be considered when changing the processing frequency of a variable frequency processor: *time overhead* and *energy overhead*. The time overhead affects the feasibility of our algorithms; that is, whether an application’s timing constraints can be met or not. We focus on time overhead first and address energy overhead later in Section 4.5.1.2.

4.5.1.1 Time Overhead and Slack Reservation The time overhead of frequency and voltage adjustment has great variations based on different architectures. For example, an AMD K6-2+ was measured to have an overhead of $400\mu s$ for changing voltage and $40\mu s$ for changing frequency [68]. Burd et al. indicate that the delay is limited by $70\mu s$ in the new ARM V4 processor based systems [14]. The frequency transition in lpRAM processors takes approximately $25\mu s$ [67]. For Intel XScale, the maximum timing overhead for changing both frequency and voltage was measured as $30\mu s$ while the StrongARM SA-1110 needs $150\mu s$ for frequency changes [72]. In our experiments, we measured that the frequency change takes $2ms$ for Transmeta Crusoe 5400 processors.

Since the time overhead of frequency adjustment may not be negligible (e.g., a few *ms*), especially for real-time systems, it needs to be taken into consideration in energy aware scheduling algorithms. One simple approach is to incorporate the time overhead of frequency changes into the WCET of each task. In the following, we propose a scheme of *slack reservation* to incorporate the time overhead of frequency changes into our energy aware scheduling algorithms.

First, we consider a model to calculate the time overhead associated with frequency changes. In this model, the time overhead consists of two components: a *constant component* that models set-up time and a *variable component* that models the time of changing frequency

with different ranges. Assuming that the variable component is linearly related to the range of frequency changes, we have:

$$O_t(f_1, f_2) = C_0 + C_1 \cdot |f_1 - f_2| \quad (4.11)$$

where C_0 and C_1 are constants. f_1 is the processing frequency before adjustment and f_2 is the processing frequency after adjustment. Notice that, the choice of $C_1 = 0$ results in a constant time overhead model.

Under the scheme of slack reservation, whenever we try to use slack to scale down processing frequency of a processing unit, we reserve enough slack for the processing unit to change its frequency back to an appropriate level in the future. That is, if the current task uses up all its allocated time and there is no additional slack, the processing frequency can be scaled up using the reserved slack and the future tasks can be executed at the appropriate frequency to meet the timing constraints. The idea is illustrated in Figure 4.14.

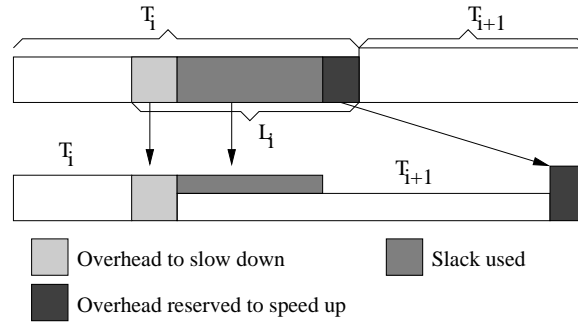


Figure 4.14: Slack reservation for incorporating frequency adjustment overhead.

From the figure, when task T_i finishes early and there are L_i units of slack, we use a portion of L_i to scale down the processing frequency and reserve enough slack for restoring frequency back to f_{max} if task T_{i+1} uses up its allocated time. The remaining slack is allocated to task T_{i+1} as additional time units to scale down the processing frequency for executing T_{i+1} .

Suppose that the current frequency for T_i is f_i and the expected frequency for T_{i+1} is f_{i+1} (to be computed). The time overhead, $O_t(f_i, f_{i+1})$, to change frequency from f_i to f_{i+1} ,

and the time overhead, $O_t(f_{i+1}, f_{max})$, to restore frequency from f_{i+1} back to f_{max} are:

$$O_t(f_i, f_{i+1}) = C_0 + C_1 \cdot |f_{i+1} - f_i| \quad (4.12)$$

$$O_t(f_{i+1}, f_{max}) = C_0 + C_1 \cdot (f_{max} - f_{i+1}) \quad (4.13)$$

Hence, f_{i+1} can be obtained after giving additional time, $(L_i - O_t(f_i, f_{i+1}) - O_t(f_{i+1}, f_{max}))$, to task T_{i+1} . That is:

$$f_{i+1} = f_{max} \cdot \frac{c_{i+1}}{c_{i+1} + (L_i - O_t(f_i, f_{i+1}) - O_t(f_{i+1}, f_{max}))} \quad (4.14)$$

For different values of L_i , f_{i+1} could be smaller than, equal to or larger than f_i . Suppose that $f_{i+1} < f_i$, Equation 4.14 will be a quadratic equation in f_{i+1} as follows:

$$2 \cdot C_1 \cdot f_{i+1}^2 + [c_{i+1} + L_i - 2 \cdot C_0 - C_1 \cdot (f_{max} + f_i)] \cdot f_{i+1} - f_{max} \cdot c_{i+1} = 0 \quad (4.15)$$

If no solution of f_{i+1} that satisfies $f_{i+1} < f_i$ is obtained from the above equation, the assumption is wrong. That is, $f_{i+1} \geq f_i$. It is possible to have $f_{i+1} = f_i$, if the slack $L_i - O_t(f_i, f_{max})$ (i.e., the overhead of $O_t(f_i, f_{i+1})$ is removed) is enough for task T_{i+1} to scale down its processing frequency to f_i . That is, if

$$f_i \geq f_{max} \cdot \frac{c_{i+1}}{c_{i+1} + L_i - O_t(f_i, f_{max})} \quad (4.16)$$

we can set $f_{i+1} = f_i$. If the above equation does not hold, we have $f_{i+1} > f_i$ and f_{i+1} can be solved as:

$$f_{i+1} = f_{max} \cdot \frac{c_{i+1}}{c_{i+1} + L_i - 2 \cdot C_0 - C_1 \cdot (f_{max} - f_i)} \quad (4.17)$$

In most cases, the reserved slack, $O_t(f_{i+1}, f_{max})$, will not be used and becomes part of the slack L_{i+1} , which can be reclaimed by next task T_{i+2} . However, after T_{i+1} finishes its execution, it is possible that the useful slack $L_{i+1} - O_t(f_{i+1}, f_{max})$ is not enough for T_{i+2} to use and the frequency computed from Equation 4.17 is larger than f_{max} . In this case, $O_t(f_{i+1}, f_{max})$ will be used to restore the frequency back to f_{max} and T_{i+2} will run at f_{max} as shown in Figure 4.15).

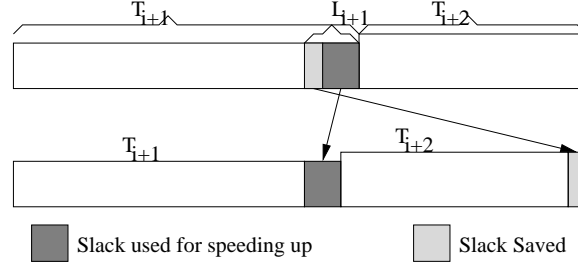


Figure 4.15: Slack is not enough for an additional frequency change.

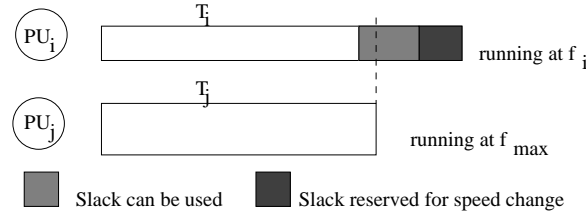


Figure 4.16: Slack sharing with frequency change overhead.

To incorporate the time overhead of frequency changes, slack sharing between processing units also needs to be modified. Referring to Figure 4.16, suppose processing unit PU_i runs at f_i and finishes early. As described in Section 4.3.2, it would share its slack with processing unit PU_j , which is running at frequency f_{max} , since PU_i supposes to finish later than PU_j . However, if there is not enough slack for PU_i to change its frequency back to f_{max} after slack sharing, we should not share the slack. Instead, PU_i needs to restore its frequency to f_{max} first.

4.5.1.2 Energy Overhead In addition to time overhead, there is also energy overhead associated with frequency changes. Suppose the energy overhead for changing frequency from f_i to f_j is $O_e(f_i, f_j)$. Assuming that the energy consumption of T_{i+1} is E_{i+1} with f_{max} and E'_{i+1} with f_{i+1} . It is not energy efficient to change the frequency from f_i to f_{i+1} for T_{i+1} if

$$O_e(f_i, f_{i+1}) + E'_{i+1} + O_e(f_{i+1}, f_{max}) > E_{i+1} + O_e(f_i, f_{max}) \quad (4.18)$$

In other words, even if the timing constraints can be met with time overhead, we may decide not to run T_{i+1} at a lower frequency if the energy overhead is larger than the energy saved by frequency changes.

4.5.2 Discrete Frequency Levels

For modern variable frequency processors, there are only a few frequency settings [34, 19, 37]. Although our energy management algorithms focus on continuous frequency as discussed earlier, they can be easily adapted to incorporate discrete frequencies.

Specifically, after calculating the processing frequency f_i^g for task T_i (see Algorithms 1 and 2), if f_i^g falls between two frequency levels ($f_l < f_i^g \leq f_{l+1}$), setting f_i^g to f_{l+1} will always guarantee that task T_i finishes on time and that the timing constraints are met. With the higher discrete frequency, not all slack will be used for task T_i and some slack will be saved for future tasks. From our experiments, by sharing slack with future tasks, the case of discrete frequencies achieves comparable energy savings as continuous frequencies as shown in Section 4.6.

Alternatively, we can emulate the single frequency execution of task T_i with two frequencies [39]. At the beginning, T_i can be executed at the lower frequency f_l and after a certain time point t_{tp} the frequency is changed to the higher frequency f_{l+1} . The value of t_{tp} can be computed as:

$$f_i \cdot D_i = f_l \cdot t_{tp} + f_{l+1} \cdot (D_i - t_{tp}) \Rightarrow \quad (4.19)$$

$$t_{tp} = \frac{f_{l+1} \cdot D_i - f_i \cdot D_i}{f_{l+1} - f_l} \quad (4.20)$$

where D_i is the total time allocated to task T_i . However, this scheme will require an additional timer for tasks to change frequencies during their execution. Furthermore, there is one more frequency change during the execution of each task that needs to be incorporated as discussed in Section 4.5.1.

4.5.3 Shared Memory Access Contention

In shared memory architectures, the data shared among processing units (e.g., the global ready task queue and the *uip* structure in Algorithms 1 and 2) must be updated in a critical section every time a task is dispatched. There will be additional waiting time due to shared memory access contention as part of the context switch.

In the worst case, one processing unit needs to wait until all other processing units finish accessing the shared data structures if all of them start the algorithm at the same time. To account for this waiting time due to shared memory access contention, when generating the canonical schedule for an application, we need to assume that the worst case will happen and every access to the shared data will incur the longest contention time. This pessimistic analysis may result in rejecting schedulable applications, but it will ensure that any execution will meet the timing constraints if an application’s canonical schedule does.

During actual executions, if one processing unit does not incur the longest contention time, some extra slack will be available and can be used to scale down the processing frequency. Thus, the algorithm that considers shared memory access contention is more conservative and, on average, more slack will be available for power management schemes.

We found that Algorithm 2 takes approximately 600 cycles without reclaiming slack in a system. For energy management that reclaims the slack, an additional 500 cycles are needed to compute the new processing frequency for the next task. These values are obtained by running the algorithm on the SimpleScalar micro-architecture simulator [15]. Here, we assume that up to six processing units are used and that each task has at most three successors. Note that, the exact number of cycles depends on the number of processing units in a system and on the number of successors each task can have in an application.

4.6 EVALUATIONS OF ENERGY MANAGEMENT SCHEMES

In this section, we evaluate our proposed energy management schemes through simulation. We implement a simulator that emulates the execution of an application at the task level on a shared memory parallel system. Before presenting the results, in what follows, we first

describe the simulation setup.

4.6.1 Simulation Setup

In our simulation, both synthetic and real applications are considered. For synthetic applications, the WCET c_i of a task T_i is generated randomly between c_{min} and c_{max} , which are the lower and upper bounds on the WCET of tasks, respectively. To emulate the run-time behavior of tasks and get different actual execution times, we define an average over worst case execution time ratio ϕ for an application. The average over worst case execution time ratio ϕ_i for task T_i is generated as a uniform distribution around ϕ . When $\phi \leq 0.5$, the lower and upper bounds for ϕ_i are 0.01 and 2ϕ ; when $\phi > 0.5$, the bounds are $2\phi - 1$ and 1. The actual execution time of task T_i follows a discretized normal distribution around $\phi_i \cdot c_i$ and the standard deviation is $0.48 \cdot (1 - \phi_i) \cdot c_i$ if $\phi_i > 0.5$ and $0.48 \cdot \phi_i \cdot c_i$ if $\alpha \leq 0.5$ (the value of 0.48 comes from discretized values of the normal distribution).

In addition, a special synthetic AND/OR-model application is considered and its dependence graph is shown in Figure 4.17. The loops in the dependence graph can be expanded as discussed in Section 3.1. The numbers associated with each loop are the maximum number of iterations paired with the probabilities of having specific numbers of iterations. If there is only one number, it is the exact number of iterations during any execution.

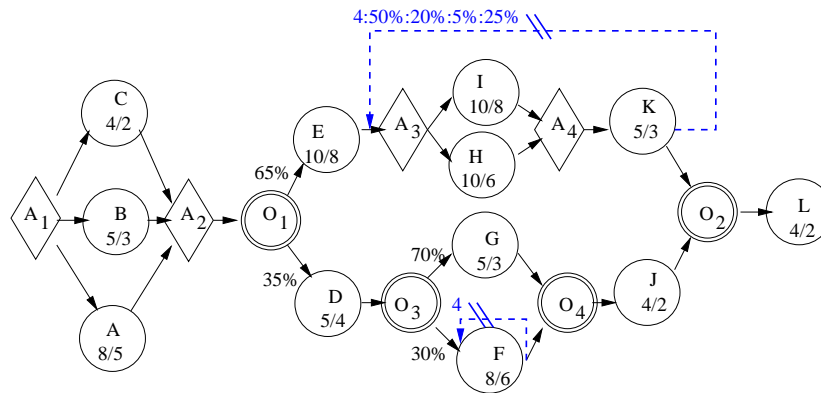
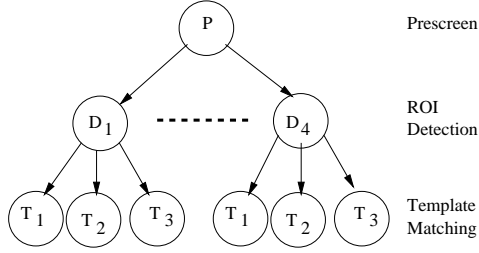


Figure 4.17: Dependence graph for a synthetic AND/OR-model application.

The first real application we considered is an automated target recognition algorithm

(ATR). ATR searches regions of interest (ROI) in one frame and tries to match some templates with each ROI [75]. The dependence graph for ATR is shown in Figure 4.18a.



	$\min(\mu s)$	$\max(\mu s)$
Prescreen	1146	1299
ROI Detection	429	748
Template 1	466	574
Template 2	466	520
Template 3	467	504

a. Dependence graph of ATR

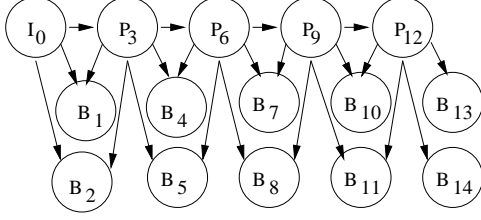
b. Execution time for tasks in ATR

Figure 4.18: The dependence graph of ATR and the execution time for the tasks in ATR. Assuming up to 4 ROIs in one frame and 3 templates.

Here, we assume that ATR can process up to four ROIs in one frame and that each ROI is compared with three different templates. If the number of ROIs is less than 4, the actual run time of the tasks corresponding to undetected ROIs (the first few ROIs) is set to 0. We instrument the ATR algorithm to record the execution times for the program sections corresponding to all the tasks. The table in Figure 4.18b shows the execution timing information about the tasks in ATR when running on a Linux machine with a Pentium-III 500MHz and 128MB memory. The execution times for tasks are the average values of ATR processing 180 consecutive frames provided by our industry partner.

Second, we consider the Berkeley real-time MPEG-1 encoder [27]. By setting the group of pictures (GOP) as 15 with the pattern of **IBBPBBPBBPBBPBB** and forcing it to encode the last frame, the dependence graph to process the frames in one GOP using decoded frame as reference is shown in Figure 4.19a.

There are three different frames. The **I** frame is the intra-frame that is encoded as a single image with no reference to any past or future frames. The **P** frame is the forward predicted frame that is encoded relative to the past reference frame. A **B** frame is a bi-directional predicted frames that is encoded relative to the past, the future or both reference



	Flower(<i>ms</i>)		Tennis(<i>ms</i>)	
	min	max	min	max
I	50	70	60	70
P	120	140	100	140
B	270	320	190	340

a. Dependence graph of MPEG-1 encoder

b. Execution time for different frames

Figure 4.19: The dependence graph and execution time to process different frames of MPEG-1 encoder; assuming that the encoding sequence is IBBPBBPBBPBBPBB, the last frame is forced to be encoded, and decoded frames are used as reference.

frames. The reference frame is either an **I** or a **B** frame. We also instrument MPEG-1 and for the *Flower-Garden* and *Tennis* movies with each having 150 frames, Figure 4.19b shows the run time information of processing different frames (the time is only for encoding and does not include I/O).

For the variable frequency processors, we first consider an ideal processor model that has continuous frequency. The maximum frequency for the ideal processors is assumed to be $f_{max} = 1$ and there is no minimum frequency limitation (i.e., $f_{min} = 0$). Moreover, the frequency-dependent active power is assumed to be $P_d = C_{ef} \cdot f^3$ with the maximum frequency-dependent active power as $P_d^{max} = C_{ef} \cdot f_{max}^3 = 1$.

Unless specified otherwise, we assume that the sleep power for each processing unit is $P_s = 0.1$ and the frequency-independent active power is $P_{ind} = 0$. Without considering turning on/off processing units dynamically due to the prohibitive cost, we have the minimum energy efficient frequency as $f_{ee} = 0$. That is, all frequencies are energy efficient. The effects of frequency-independent active power P_{ind} on the performance of energy management schemes are addressed separately in Section 4.6.6.

In addition, two real processor models are considered. In the Transmeta model [37], there are 16 frequency/voltage settings as shown in Table 4.2. The second power configuration that we considered is the Intel XScale model [34] with the frequency/voltage settings as shown in Table 4.3. From the tables, we can see that the Intel XScale model has a wider

frequency/voltage range than the Transmeta model but fewer frequency/voltage levels.

Table 4.2: Frequency/voltage settings for Transmeta 5400

$f(MHz)$	700	666	633	600	566	533	500	466
$V_{dd}(V)$	1.65	1.65	1.60	1.60	1.55	1.55	1.50	1.50
$f(MHz)$	433	400	366	333	300	266	233	200
$V_{dd}(V)$	1.45	1.40	1.35	1.30	1.25	1.20	1.15	1.10

Table 4.3: Frequency/voltage setting for Intel XScale processors

$f(MHz)$	1000	800	600	400	150
$V_{dd}(V)$	1.80	1.60	1.30	1.00	0.75

Note that the frequencies and voltages do not obey a linear relation in either model, which is different from the ideal processors. Therefore, the frequency-dependent power for these two real processor models is assumed to be $P_d = C_{ef} \cdot f \cdot V_{dd}^2$. Again, we assume that the sleep power of one processing unit consumes 10% of the maximum frequency-dependent power P_d^{max} .

In our simulations, we consider the following schemes: the no power management (NPM), the greedy static power management (G-SPM), the uniform static power management (U-SPM), the static power management with parallelism (SPM-P), the shifting/sharing slack reclamation scheme (S/SSR), the static speculation scheme (SS), the adaptive speculation scheme (AS) and one clairvoyant scheme (CLV). Following the idea of the optimal scheduling technique to minimize energy consumption for uniprocessor systems [39], CLV uses the actual execution time of tasks to generate a schedule and to compute a single frequency for all tasks.

We vary a number of parameters in our simulations, such as the *overhead* of frequency changes, the *number of frequency levels*, the *laxity over deadline ratio (LDR)*, the *variability* (ϕ) of an application's computation requirement and the minimum energy efficient frequency f_{ee} to see how they affect the energy savings for our energy management schemes. The *laxity* is defined as the difference between an application's worst case execution time and its deadline D . *LDR* is defined as $LDR = \frac{laxity}{D}$ and indicates the amount of *static slack* in a system.

The *variability* of an application’s computation requirement, denoted by ϕ , indicates the average amount of *dynamic slack* generated during the actual execution of the application.

4.6.2 Effects of Frequency Change Overhead

We first consider synthetic tasks with $c_{min} = 1$ and $c_{max} = 50$. Recall that the time overhead of frequency change is modeled as a linear function of the range of frequency changes (see Section 4.5.1). To observe how the time overhead affects the performance of our proposed energy management schemes in terms of energy savings, we set in the simulations the constant component of the time overhead (C_0) to different values relative to the smallest task’s WCET. The maximum variable component corresponds to changing frequency between f_{max} and f_{min} and equals C_1 times the smallest task’s WCET, where C_1 is set to different values from 0 to 1.

Figures 4.20ab shows the effects of different values of C_0 and C_1 (i.e., different frequency change overheads) on the energy savings of S/SSR. We consider an independent task set with 100 tasks and a dependent task set with 20 tasks that are executed on a system with two processing units. In this simulation, we set $LDR = 0$ and $\phi = 0.5$. The normalized energy consumption is reported with the energy consumed by U-SPM as a basis. In the results, each data point corresponds to an average of 1000 executions.

From the figures, for independent tasks, there is a 6% difference in energy consumption between the case of maximum overhead and the case of no overhead. For dependent tasks, the difference is 12%. There is a big jump between the case with no overhead and that with minimal overhead for dependent tasks. The reason is that the gaps in the middle of a schedule run at f_{min} when no frequency change overhead is considered; however, when there is frequency change overhead, the gaps run at f_{max} to ensure that future tasks finish on time.

For the real variable frequency processors, the time overhead of frequency changes has great variations from a few micro-seconds to a few milli-seconds (see Section 4.5.1). With technology advancements, the frequency change overhead is expected to decrease. Unless specified otherwise, in what follows, we assume that changing the frequency (and the corresponding supply voltage) once takes $5\mu s$.

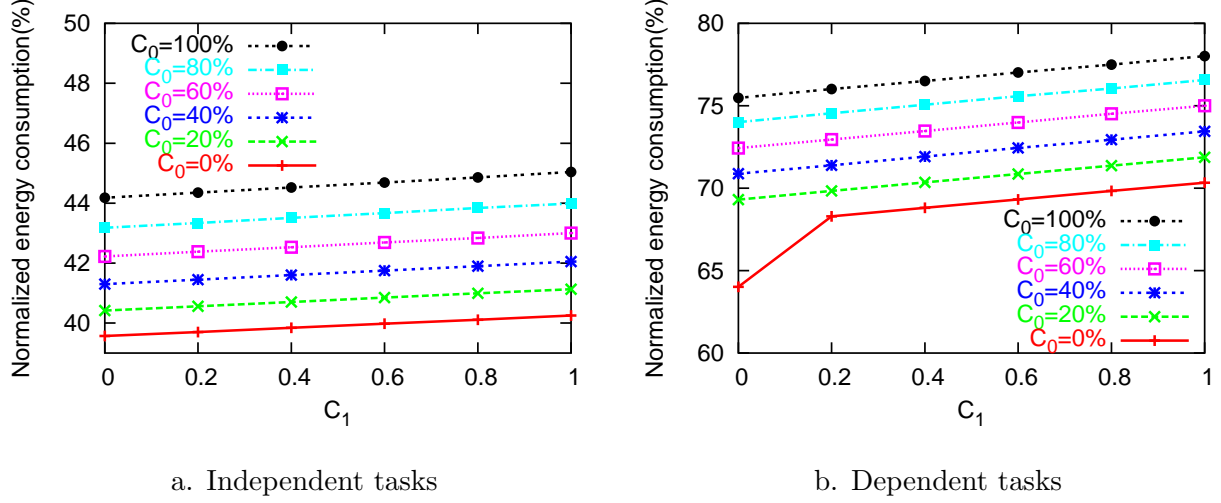


Figure 4.20: The effects of frequency change overhead on energy savings.

4.6.3 Effects of Discrete Frequency Levels

To see how discrete frequency levels affect the performance of energy management schemes on energy savings, we consider processors that have different numbers of frequency levels between $f_{min} = 200MHz$ and $f_{max} = 700MHz$ (the frequency numbers are taken from Transmeta 5400 [37]). Figure 4.21a shows the normalized energy consumption of S/SSR when the same synthetic task sets used in the last section are executed on a system with two processing units. To separate the effects of discrete frequency levels and frequency change overhead, no frequency change overhead is considered in this section.

The frequency levels are uniformly distributed at the same increment between two discrete frequency levels and ' ∞ ' corresponds to the case of continuous frequency. From the figure, we can see that energy consumption of S/SSR with continuous frequency is not always less than that of discrete frequencies, and that more frequency levels do not guarantee less energy consumption. The reason is that, with discrete frequencies, processors set their frequency to the next higher discrete level, which saves some slack for future tasks. When sharing the slack with future tasks, the energy consumption with discrete frequencies may be less than that with continuous frequency, and a few frequency levels may be better than many frequency levels. Moreover, 4 or 6 frequency levels are sufficient to achieve almost the

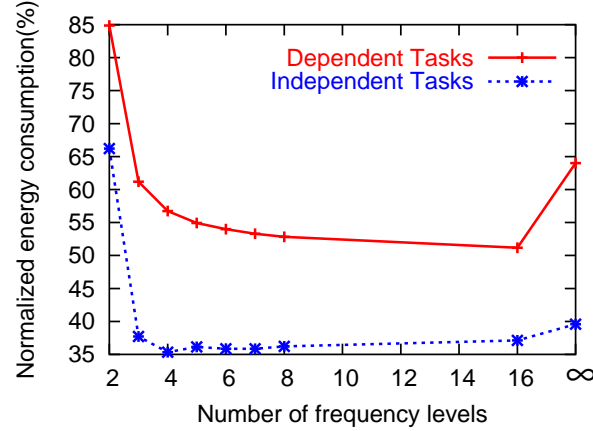


Figure 4.21: The effects of discrete frequencies on energy savings.

same energy savings as the case of continuous frequency, which is the same observation as reported in [17] for uniprocessor with periodic tasks.

4.6.4 Energy Savings of SPM-P

We compare the energy consumed by our new scheme SPM-P with G-SPM and U-SPM in this section. We consider synthetic applications that consist of randomly generated dependent tasks. Different numbers of tasks (from 7 to 90) in an application have been tested, since the nature of the results is more or less the same, we only present the results for an application that consists of 50 dependent tasks. Moreover, we set $c_{min} = 10$ and $c_{max} = 100$.

Since LDR determines the amount of static slack in a system, Figure 4.22 shows the normalized energy consumption for different static power management schemes as a function of LDR when the application is executed on four and eight processors, respectively. Here, we consider the Intel XScale processor model and normalized energy consumption is reported with the energy consumed by NPM being used as a basis.

Notice that the length of an application's canonical schedule is fixed when it is executed on a specific system. To get different amount of static slack in a system (i.e., different values of LDR), we vary the deadline of an application. From Figure 4.22a, we can see that SPM-P performs the best in terms of energy savings when compared with G-SPM and U-SPM. On

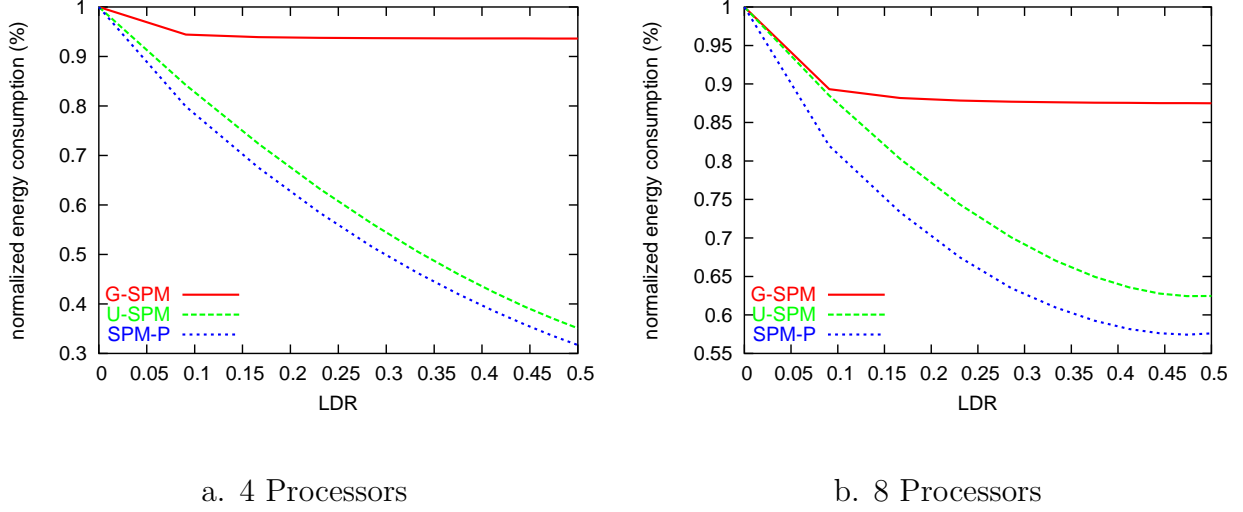


Figure 4.22: The normalized energy vs. LDR for different SPMs.

increasing the number of processors to 8, Figure 4.22b shows that SPM-P experiences more energy savings compared with G-SPM and U-SPM. The reason is that the variations of parallelism in the schedule become larger for 8 processors and SPM-P is able to exploit these variations by allocating more slack to schedule sections that have higher levels of parallelism. It is clear that more energy savings can be obtained with larger LDR (i.e., more static slack). Finally, compared with U-SPM, SPM-P saves around 5% more energy.

Although not shown, the total energy consumed in case of 8 processors is greater than what is consumed by 4 processors. It comes from the fact that total idle time typically increases for larger number of processors due to more synchronization being needed on more processors (recall that an idle processing unit consumes the sleep power which is assume to be $P_s = 0.1$).

4.6.5 Energy Savings of S/SSR and Speculative Schemes

4.6.5.1 Trace Based Simulations for AND-model Applications When using the traces from ATR and MPEG-1, we vary the number of processors in the simulations to show the effectiveness of our shifted/shared slack reclamation schemes on different systems. Note that the maximum parallelism for Berkeley MPEG-1 encoder is 3 for one GOP. The timing

information about the traces were shown in Section 4.6.1. The Intel XScale processor model is used. Notice that, both ATR and MPEG-1 are AND-model applications. Thus, S/SSR acts the same as shared slack reclamation (SSR), a special case of S/SSR. The energy savings of S/SSR compared with U-SPM are shown in Table 4.4.

Table 4.4: Energy savings vs. U-SPM using trace data

	ATR			MPEG-1 Encoder			
	2-Proc	3-Proc	4-Proc	Flower		Tennis	
				2-Proc	3-Proc	2-Proc	3-Proc
S/SSR	26.35%	38.65%	41.66%	17.42%	16.53%	25.16%	23.77%
CLV	58.83%	54.71%	52.14%	24.11%	26.43%	35.07%	36.92%

From the table, we can see that there is more energy savings for *Tennis* than *Flower-Garden* from MPEG-1 encoder because the encoding time for *Tennis* varies more than *Flower-Garden* (see Figure 4.19b). CLV gets 7% to 32% more energy savings than S/SSR.

4.6.5.2 Synthetic AND/OR-model Applications Notice that, when using the trace data, the amount of dynamic slack in the traces is fixed. To get different amounts of dynamic slack and examine their effects on the energy savings of S/SSR and speculative schemes, we consider the synthetic AND/OR-model application as shown in Figure 4.17. When the synthetic application is executed on a dual-processor system with $LDR = 0.2$, the normalized energy consumption for the energy management schemes is shown in Figure 4.23 as a function of ϕ with energy consumed by U-SPM as a basis. Here, we consider two real processor models as mentioned in Section 4.6.1: Transmeta [37] and Intel XScale [34], which have 16 and 5 frequency levels, respectively.

From the figure, we can see that more energy savings are obtained for large values of ϕ since they indicates more dynamic slack. Moreover, the performance of S/SSR on energy savings is as good as that of static speculation (SS) and adaptive speculation (AS) schemes.

It is expected that the speculative schemes perform better than the S/SSR scheme. The reason is that, typically, S/SSR tends to allocate all the available slack to the current task and executes the task at the slowest possible frequency. Consequently, future tasks may need to

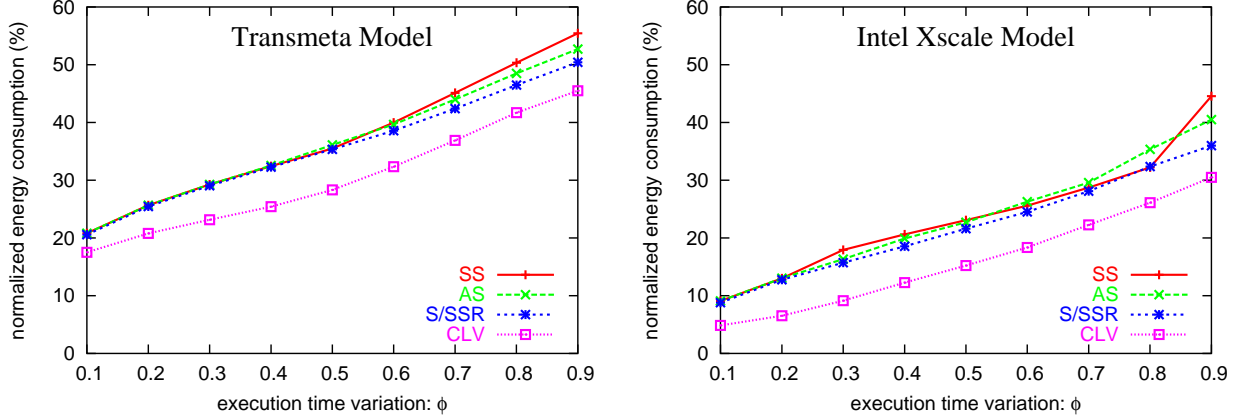


Figure 4.23: The normalized energy vs. ϕ for synthetic application running on a dual-processor system with $LDR = 0.2$.

run at higher frequencies and consume more energy. However, when the minimum frequency is bounded by f_{min} , it prevents S/SSR from using all available slack at the very beginning and forces some slack to be saved for future tasks. Moreover, as shown in Section 4.6.3, fewer frequency levels can also help to prevent the S/SSR scheme from using slack early by decreasing the probability of frequency changes: the closer the frequencies are to each other, the higher the probability is for smaller amount of slack causing a frequency change. As a result, the greediness of the S/SSR scheme is moderated by a higher f_{min} and/or fewer frequency levels.

4.6.6 Effects of The Minimum Energy Efficient Frequency

From the last section, we can see that the minimum frequency has great effects on the performance of energy management schemes. So far we have ignored the frequency-independent active power P_{ind} , which could impose a minimum energy efficient frequency f_{ee} on energy management as discussed in Section 3.2. In this section, we study the effects of frequency-independent active power on energy management by setting f_{ee} with different values (i.e., different frequency-independent active powers).

With fixed $f_{max} = 1GHz$, we set the factor $\frac{f_{max}}{f_{ee}}$ to different values to get different f_{ee} ,

which correspond to different frequency-independent powers P_{ind} . Without loss of generality, we assume that the minimum frequency limitation is bounded by f_{ee} . Corresponding to Intel XScale and Transmeta processor models, 5 and 16 frequency levels are considered, which are equally distributed between f_{max} and f_{ee} . Figure 4.24 shows the normalized energy consumption of different schemes for ATR running on a dual-processor system with $LDR = 0.2$.

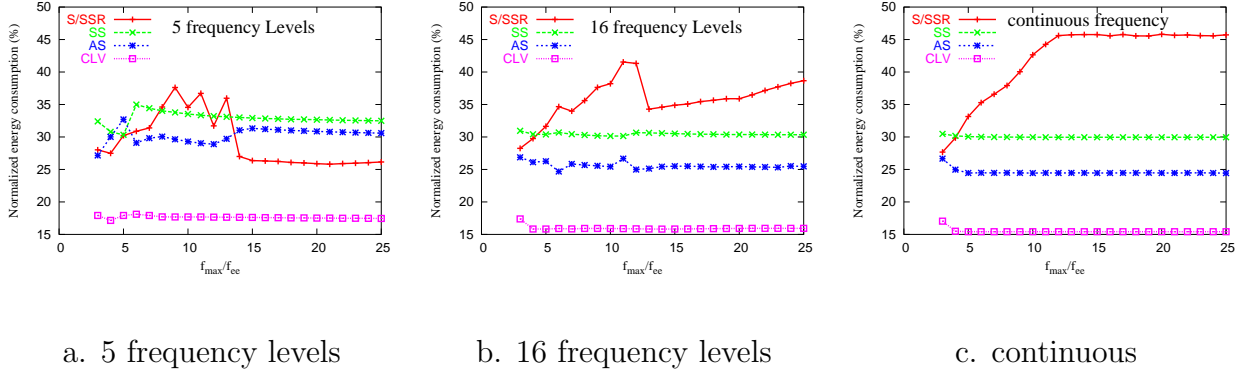


Figure 4.24: The normalized energy vs. $\frac{f_{max}}{f_{min}}$ for ATR running on a dual-processor system with $LDR = 0.2$.

As shown in Figure 4.24a, When there are 5 frequency levels, S/SSR consumes comparable amount of energy as the speculative schemes even when $f_{ee} = 40MHz$ (i.e., $\frac{f_{max}}{f_{ee}} = 25$), which coincides with our previous observation that a few frequency levels are good enough for the S/SSR scheme. For the case of 16 frequency levels as shown in Figure 4.24b, S/SSR becomes worse than speculative schemes as expected when $f_{ee} \leq 200MHz$ (i.e., $\frac{f_{max}}{f_{ee}} \geq 5$). Notice that S/SSR does not always become worse with decreased f_{ee} (i.e., increased $\frac{f_{max}}{f_{ee}}$) for the cases of 5 and 16 frequency levels. The non-monotonic changes in the performance of S/SSR are a direct result of the quantized frequency levels. When we use continuous frequencies between f_{max} and f_{ee} as shown in Figure 4.24c, the energy consumption of S/SSR increases monotonically with decreased f_{ee} . Overall, we can see that the performance of speculative schemes is 10% to 15% worse than CLV scheme. For higher LDR (e.g., $LDR = 0.6$), similar results are obtained.

4.7 THEORETICAL BOUNDS: HOW MUCH BETTER CAN WE DO?

From the evaluations, we can see that the amount of slack, both static and dynamic, determines how much we can scale down the processing frequency for executing an application, and thus the amount of energy that we can save by deploying energy management schemes. If there is no slack, an application needs to always run at the maximum frequency and no energy saving can be obtained.

So far we did not consider turning processing units off due to the prohibitive overhead. In order to answer the question of “*how much better we can do*”, we study the effects of turning processing units off in this section. Moreover, to obtain the upper bound on the energy savings that energy management schemes can achieve, no overhead of turning on/off processing units is considered.

For simplicity, we assume that applications are *fully parallel*, which means that when an application needs L time units on one processing unit, it only takes $\frac{L}{N}$ time units when it is executed on N processing units (i.e., perfect load balancing is assumed). Moreover, we assume that the *actual running time* of an application is known a priori and continuous frequency is considered. Suppose that an application would *actually* take \hat{L} time units on one processing unit at the maximum frequency f_{max} . The *actual system load* is defined as $\hat{\sigma} = \frac{\hat{L}}{D}$, where D is the application’s deadline. If the *system load* is $\sigma = \frac{L}{D}$ and the average over worst case execution time is ϕ , on average, we have $\hat{\sigma} = \phi\sigma$.

When the application is executed on a system consisting of N processing units, the *effective system load* is defined as $\hat{\sigma}_N = \frac{\hat{\sigma}}{N}$. Thus, $\hat{\sigma}$ can be up to N and $\hat{\sigma}_N$ is always bounded by 1.

For a certain system load $\hat{\sigma}$, the number of processing unit needed is at least $\lceil \hat{\sigma} \rceil$. More processing units may be used for executing the application at lower frequencies, but using fewer processing units is not feasible since it will require the processing units to run at frequencies higher than f_{max} .

Recall that the power of one processing unit is modeled as (see Section 3.2; the equation is repeated for convenience):

$$P = P_s + \hbar(P_{ind} + P_d) = P_s + \hbar(P_{ind} + C_{ef}f^m) \quad (4.21)$$

Although the application only takes $\frac{\hat{L}}{N}$ time units when it runs on N processing units at frequency f_{max} , all N processing units will consume the maximum power P_{max} for all the time if no power management (NPM) is deployed. Here, we have $P_{max} = P_s + P_{ind} + C_{ef} f_{max}^m = (\alpha + \beta + 1)P_d^{max}$ (recall that the maximum frequency-dependent active power is defined as $P_d^{max} = C_{ef} f_{max}^m$, the sleep power $P_s = \alpha P_d^{max}$ and the frequency-independent active power $P_{ind} = \beta P_d^{max}$). Therefore, the total energy consumption for NPM within one application frame D will be:

$$E_{NPM}(N, f_{max}, D) = N \cdot P_{max} \cdot D = N \cdot (\alpha + \beta + 1) P_d^{max} \cdot D \quad (4.22)$$

Under energy management, we can scale down the processing frequency of the processing units and/or turn off some processing units for energy savings. Due to the sleep power P_s and/or frequency-independent active power P_{ind} , there is an energy efficient frequency $f_{ee} = \sqrt[m]{\frac{\alpha + \beta}{m - 1}} = \kappa \cdot f_{max}$ (see Section 3.2). Therefore, when the actual system load is low (i.e., $\hat{\sigma} \leq N \cdot \kappa$), we should only deploy $X = \lceil \frac{\hat{\sigma}}{\kappa} \rceil$ processing units and turn the remaining processing units off. With perfect load balancing, each processing unit will run the application for $\frac{\hat{L} \cdot f_{max}}{X \cdot f_{ee}} = \frac{\hat{L}}{X \cdot \kappa}$ time units at frequency f_{ee} . The energy consumption is:

$$\begin{aligned} E_{PM}(X, f_{ee}, \frac{\hat{L}}{X \cdot \kappa}) &= X \cdot (P_s + P_{ind} + C_{ef} f_{ee}^m) \cdot \frac{\hat{L}}{X \cdot \kappa} \\ &= (\alpha + \beta + \kappa^m) P_d^{max} \cdot \frac{\hat{L}}{\kappa} \end{aligned} \quad (4.23)$$

Thus, compared with NPM, the amount of energy saved by energy management is the difference between $E_{NPM}(N, f_{max}, D)$ and $E_{PM}(X, f_{ee}, \frac{\hat{L}}{X \cdot \kappa})$. Define the *normalized energy savings (NES)* as the amount of energy saved by energy management over the amount of energy consumed by NPM. We have:

$$NES = \frac{E_{NPM}(N, f_{max}, D) - E_{PM}(X, f_{ee}, \frac{\hat{L}}{X \cdot \kappa})}{E_{NPM}(N, f_{max}, D)} = 1 - \frac{\alpha + \beta + \kappa^m}{\alpha + \beta + 1} \cdot \frac{\hat{\sigma}}{N \cdot \kappa} \quad (4.24)$$

When the actual system load $\hat{\sigma} > N \cdot \kappa$, all N processing units will be deployed for energy efficiency. The processing frequency for all processing units and the energy consumption are:

$$f_N = \frac{L}{N \cdot D} f_{max} = \hat{\sigma}_N f_{max} \quad (4.25)$$

$$\begin{aligned} E_{PM}(N, f_N, D) &= N \cdot (P_s + P_{ind} + C_{ef} f_N^m) \cdot D \\ &= N \cdot (\alpha + \beta + \hat{\sigma}_N^m) P_d^{max} \cdot D \end{aligned} \quad (4.26)$$

Therefore, compared with NPM, the normalized energy savings is:

$$NES = \frac{E_{NPM}(N, f_{max}, D) - E_{PM}(N, f_N, D)}{E_{NPM}(N, f_{max}, D)} = 1 - \frac{\alpha + \beta + \hat{\sigma}_N^m}{\alpha + \beta + 1} \quad (4.27)$$

As α , β , κ , m and N are system dependent parameters, the normalized energy savings will depend on actual system load $\hat{\sigma}$.

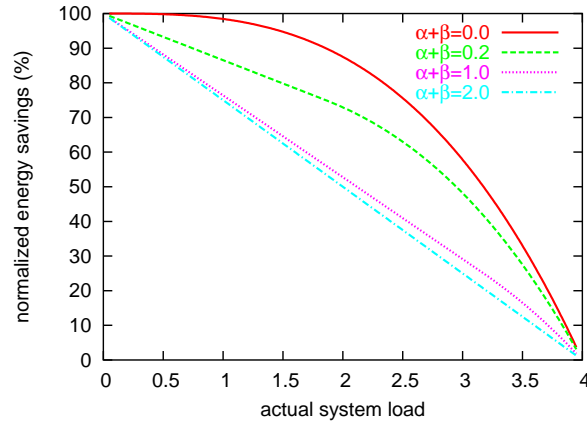


Figure 4.25: The theoretical bounds on energy savings.

Suppose that $N = 4$ and $m = 3$, Figure 4.25 shows the maximum normalized energy savings for different actual system loads that can be achieved by energy management schemes. Notice that, the overhead of turning on/off a processing unit is assumed to be zero to get the maximum bound. That is, α and β have the same effect on energy consumption as well as the minimum energy efficient frequency. For simplicity, we show the theoretical bounds for different values of $\alpha + \beta$.

From the figure, we can see that, the lower the actual system load $\hat{\sigma}$ is, the more slack there is in a system and the higher the normalized energy savings will be. When $\alpha + \beta = 0$ (i.e., there is no sleep power and frequency-independent power), energy management through voltage scaling gets the highest energy savings for specific actual system load. Notice that, with $m = 3$, the minimum energy efficient frequency is f_{max} when $\alpha + \beta \geq 2$. That is, for any workload, it is most energy efficient to execute an application at the maximum processing frequency and then turn off the processing units when they are idle. Thus, for the case of $\alpha + \beta = 2$, the maximum energy savings has a linear relation with the actual system load.

Therefore, sleep power and frequency independent power have great effects on the energy savings that can be obtained by energy management schemes. The smaller they are, the more energy savings energy management schemes can potentially obtain. For example, when $\hat{\sigma} = 2$ (i.e., half of the maximum system load for four processing units), energy management can save up to 85% for the case of $\alpha + \beta = 0$ while only 50% for the case of $\alpha + \beta = 2$.

4.8 CHAPTER SUMMARY

In this chapter, we proposed several energy management schemes for frame-based parallel real-time applications running on shared memory systems with multiple processing units. These schemes include both static and dynamic energy management to reclaim static and dynamic slack, respectively. We considered AND-mode applications, where all tasks of an application are always executed, as well as AND/OR-model applications that have more than one execution path and only a subset of tasks will be executed during any specific execution instance.

As a first step, we addressed the importance of task priority assignment in parallel real-time systems. We illustrated the anomaly of traditional list scheduling with longest task first (LTF) heuristic. We proved that, without reclaiming slack, fixed-priority list scheduling can meet an application's timing constraints if the timing constraints can be met in the application's canonical execution, in which every task is assumed to use its worst case execution time (WCET). We then proposed the fixed-priority list scheduling for energy management in parallel real-time systems.

While the greedy static power management (G-SPM) gives all static slack to the first or last task running on each processing unit, the uniform static power management (U-SPM) distributes static slack over the schedule evenly. Although U-SPM is optimal for uniprocessor systems, it is not optimal for parallel systems due to different levels of parallelism for different sections in a schedule. With the observation that it is more energy efficient to allocate more slack to schedule sections that have higher levels of parallelism, a static power management scheme with parallelism (SPM-P) is proposed. SPM-P allocates static slack to different sections of a schedule based on their levels of parallelism. Simulation results show that, compared to U-SPM, SPM-P can save up to 5% more energy. However, the actual energy savings will depend on the parallelism variations in a schedule.

When considering *dynamic slack*, the simple *greedy slack reclamation (GSR)* scheme, which allocates any dynamic slack to the next task to be executed and has been demonstrated to be effective for uniprocessor systems [6]. However, we showed the GSR is not feasible for parallel systems and may result in missed deadlines. Based on a global fixed priority list scheduling strategy, the *shifted/shared slack reclamation (S/SSR)* schemes are proposed. Simulation results show that S/SSR can get up to 50% energy savings compared with static power management when the average over worst case execution time of an application is around 50%. With the capability of automatically balancing actual workload among processing units through global scheduling, the proposed schemes have been shown to save more energy than partition scheduling where each processing unit reclaims slack for energy management individually [94, 95].

Considering the overhead of changing frequency/voltage and observing that a new frequency is calculated for each task under the S/SSR scheme, the *speculation schemes* are proposed by exploring the statistical timing information about applications. The speculation schemes intend to save more energy by reducing the number of frequency/voltage changes and the related overhead. However, our simulation results show that the speculation schemes consume comparable energy with S/SSR when there is a minimal frequency limitation and/or only a few frequency/voltage levels exist in a system.

Some practical issues for energy management are also addressed, such as the overhead of frequency/voltage changes, discrete frequency/voltage levels and shared memory access

contention. A *slack reservation* scheme has been proposed to incorporate the overhead of frequency/voltage changes into the energy management algorithms. From our experiments, it is shown that, when frequency/voltage scaling overhead is relatively small compared to the size of real-time tasks, the effects of such overhead is not significant [95]. In addition, a few (e.g., 4 to 6) frequency/voltage levels are as effective as continuous frequency/voltage for energy management on energy savings, which coincides with the observation as reported in [17] for uniprocessor with periodic tasks. When shared memory contention is considered, our energy aware scheduling algorithms become more conservative and may reject applications that are schedulable. However, for applications that have a feasible schedule under our energy aware scheduling algorithms, more energy savings are obtained.

Moreover, we addressed the question of “*what is the best we can do*” using energy management and provided some theoretic bounds on energy savings. Clearly, the amount of energy that could be saved depends on the total amount of slack, both static and dynamic, in a system. For systems with 100% load, there is no slack and no energy saving can be obtained through energy management. When a system is not fully loaded, the lower the system load is, the more slack there is and the higher the energy savings can be obtained through energy management.

5.0 ENERGY EFFICIENT FAULT TOLERANCE

For mission critical real-time applications, such as satellite and surveillance systems, higher levels of reliability are also desired in addition to lower levels of energy consumption. Although fault tolerance through redundancy [69, 70] and energy management through slack reclamation [62, 66, 76, 85, 88] have been extensively explored separately, there is less work focusing on the combination of energy and reliability management.

In this chapter, we address the problem of energy efficient fault tolerance for real-time systems. *Performability* has been defined as the probability of finishing an application correctly within the application's deadline in the presence of faults [42]. In this work, performability is also used to refer to the number of faults that can be tolerated within an application's deadline. Therefore, larger performabilities imply higher levels of reliability and smaller performabilities correspond lower levels of reliability.

First, exploring slack time as temporal redundancy, we discuss *energy efficient roll-back recovery* schemes. The optimal number of checkpoints for minimizing energy consumption or maximizing performability is explored (Section 5.1). Then, we focus on an *optimistic N-modular redundancy (ONMR)* scheme, which reduces the processing frequency for some processing units in a N -modular redundant system for energy savings provided that these processing units can speed up the computation whenever needed to meet the deadlines (Section 5.2). For a system that consists of a fixed number of processing units, we also look at the configuration problems and propose a framework to find the optimal redundant configuration for either minimizing the energy consumption or maximizing system performability (Section 5.3). Finally, we discuss the interplay of energy and performability management and address the trade-off between energy consumption and performability in a system. The effects of energy management on fault rates are also considered and an exponential fault rate

model is proposed based on previous published data (Section 5.4).

5.1 ENERGY EFFICIENT ROLL-BACK RECOVERY

We have discussed the management schemes that explore slack in real-time systems for energy savings in Chapter 4. In this section, slack time will be explored as temporal redundancy to increase system performability. Notice that, for a reliable real-time system, the performability generally corresponds to the worst case scenario (e.g., the probability of finishing an application correctly or the maximum number of faults that can be tolerated within the application's deadline in the worst case). Therefore, only static slack is considered for temporal redundancy in this work. The usage of dynamic slack, which depends on the run-time behavior of an application, for performability will be explored in our future work.

As a first step, we consider single task applications. However, the schemes considered can be easily extended to applications that consist of multiple tasks by enforcing an individual deadline for each task after allocating the static slack. Suppose that an application has WCET as L and deadline D . The amount of static slack available is $D-L$ and the system load is defined as $\sigma = \frac{L}{D}$. We will assume in this chapter that replicated execution of an application on multiple processing units is used to detect faults (see Section 3.3). However, for simplicity, we assume in this section that the application is executed only on two processing units (i.e., a Duplex system).

We first consider the simple case of re-executing an application to recover from transient faults (Section 5.1.1). The concept of *pessimism level* for the number of expected faults is proposed. Then checkpointing is considered for efficient usage of slack (Section 5.1.2) and the optimal number of checkpoints that maximizes performability with limited energy budget is explored (Section 5.1.3). The optimal number of checkpoints that minimizes energy consumption with a given performability goal is also explored (Section 5.1.4). We further evaluate the effectiveness of checkpointing by comparing the energy consumption and performability of a duplex system with checkpoints and a triple modular redundancy (TMR) system that uses one more processing unit (Section 5.1.5).

5.1.1 Simple Scheme of Re-execution (Retry)

In a retry scheme, an application is re-executed *as a whole* when the results on two processing units in a duplex system are not identical. For an application with WCET L , the maximum number of recoveries b_{max} that can be scheduled at the maximum frequency f_{max} within its deadline D is:

$$b_{max} = \left\lfloor \frac{D}{L} \right\rfloor - 1 \quad (5.1)$$

Here, the overhead of reloading an application for re-execution is assumed to be incorporated into the application's WCET, L .

5.1.1.1 Pessimism Level: The Number of Expected Faults When only b ($\leq b_{max}$) recoveries are needed, the amount of remaining static slack is $D - (b+1)L$. This slack can be used to scale down the processing frequency of the original execution as well as recoveries of an application. Considering the probability of recoveries being invoked (e.g., the i^{th} recovery is invoked only if an application and all the previous $i - 1$ recoveries fail), it may be more energy efficient to execute the original execution with a lower frequency while the recoveries are executed at higher frequencies. For simplicity, in this work, the frequency to execute any recovery is *either* the maximum frequency f_{max} *or* the same frequency of the original execution.

Assuming that no faults will occur (i.e., being optimistic), all the slack $D - (b+1)L$ can be used to scale down the processing frequency of the application's original execution. The recoveries are executed at the maximum frequency f_{max} if needed. Alternatively, assuming that all recoveries of an application are needed (i.e., being pessimistic), we can scale down the processing frequency for the original execution as well as all the recoveries of an application to minimize the expected energy consumption. We define the *pessimism level* b_e ($\leq b$) as the number of recoveries that are expected to be invoked (i.e., the number of faults expected to occur). Therefore, optimistic analysis corresponds to $b_e = 0$ and pessimistic analysis corresponds to $b_e = b$. When computing the expected energy consumption, we assume that the slack is used to scale down the processing frequency of the original execution and the first b_e ($\leq b$) recoveries of an application.

For example, as illustrated in Figure 5.1, if an application has $L = 2$ and $D = 6$, the maximum number of recoveries that can be scheduled within D at the maximum frequency $f_{max} = 1$ is two (Figure 5.1a). In this case, there is no additional slack and the application is executed at the maximum frequency $f_{max} = 1$ during its original execution as well as its recoveries if needed. However, if only one recovery is needed, there will be $6 - 2 \cdot 2 = 2$ units of slack. Being optimistic, the slack should be used to scale down the processing frequency of the application's original execution, which will be executed at the frequency of $\frac{1}{2}$, while the recovery is executed at $f_{max} = 1$ as shown in Figure 5.1b. If we are pessimistic and expect that there is a fault during the original execution of the application, the recovery will be needed and the slack should be used to scale down the processing frequency for both the application's original execution and its recovery (Figure 5.1c). In the figures, each task box represents replicated execution of the application on two processing units.

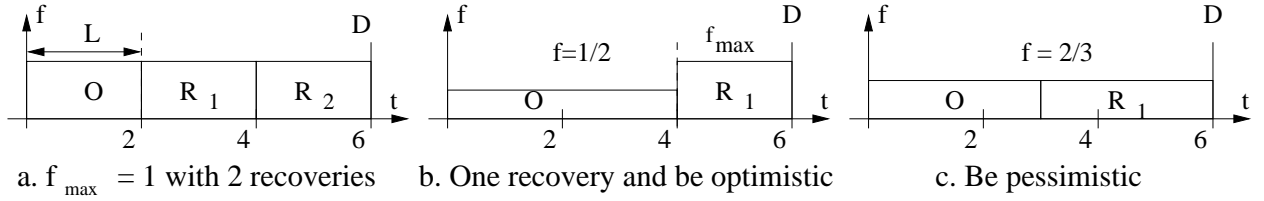


Figure 5.1: The retry scheme for an application with $L = 2$ and $D = 6$. Here, O stands for the original execution and R_i ($i = 1, 2$) is for the i^{th} recovery.

5.1.1.2 Performability of Retry Scheme Suppose that b ($\leq b_{max}$) recoveries are scheduled to achieve a certain level of performability for an application under the retry scheme. With b_e -pessimism ($b_e \leq b$), the amount of time allocated to the application's original execution or any of its first b_e recoveries is:

$$t_{org} = \frac{D - (b - b_e)L}{b_e + 1} \quad (5.2)$$

With a Poisson distribution of faults and average fault arrival rate λ , the probability of having fault(s) in one processing unit during the application's original execution or any execution

of its first b_e recoveries is $1 - e^{-\lambda t_{org}}$. Therefore, the probability of the application's original execution or any execution of its first b_e recoveries having fault(s) on a duplex system is:

$$\rho_{org} = 1 - e^{-2\lambda t_{org}} \quad (5.3)$$

Notice that each of the remaining recoveries takes L time units at the maximum frequency $f_{max} = 1$. Similarly, the probability of any remaining recovery having fault(s) on a duplex system is:

$$\rho_{rem} = 1 - e^{-2\lambda L} \quad (5.4)$$

Thus, the performability (i.e., the probability of an application being executed correctly within its deadline) is:

$$R_{retry} = (1 - \rho_{org}^{b_e+1}) + \rho_{org}^{b_e+1}(1 - \rho_{rem}^{b-b_e}) \quad (5.5)$$

where the first part is the probability of the application being executed correctly within its original execution and the first b_e recoveries; the second part is the probability of having fault(s) during its original execution and all the first b_e recoveries while one of the remaining recoveries getting correct results.

5.1.1.3 Expected Energy Consumption of Retry Scheme With an application's original execution and any of its first b_e recoveries being allocated t_{org} time units (see Equation 5.2), the frequency, at which the application's original execution and the first b_e recoveries will be executed, is:

$$f_{org} = \frac{(b_e + 1)L}{D - (b - b_e)L} \quad (5.6)$$

Therefore, the energy consumed by the application's original execution or any of its first b_e recoveries on a duplex system is:

$$E_{org} = 2(P_s + P_{ind} + C_{ef} f_{org}^m) t_{org} \quad (5.7)$$

Since the remaining recoveries are executed at f_{max} , the energy consumption for executing any of the remaining recoveries on a duplex system is:

$$E_{rem} = 2(P_s + P_{ind} + C_{ef}f_{max}^m)L \quad (5.8)$$

For the retry scheme, the original execution of an application needs to be performed under all circumstances. The probability of the first recovery being executed is the probability of having fault(s) during the original execution, which is ρ_{org} as shown in Equation 5.3. Similarly, the probability of the i^{th} ($i \leq b$) recovery being executed is the probability of having fault(s) during the original execution as well as every previous recovery execution, which is given as follows:

$$Pr_i = \begin{cases} \rho_{org}^i & 1 \leq i \leq b_e \\ \rho_{org}^{b_e+1} \cdot \rho_{rem}^{i-b_e-1} & b_e < i \leq b \end{cases} \quad (5.9)$$

Thus, the *expected energy consumption* for executing the application is:

$$E_{exp} = E_{org}(1 + \sum_{i=1}^{b_e} Pr_i) + E_{rem} \sum_{i=b_e+1}^b Pr_i \quad (5.10)$$

5.1.2 Checkpointing and Its Applicability

Notice that, the retry scheme is not applicable when system load is more than 50%. For large applications, re-execution takes significant amount of slack time. To efficiently use slack time, checkpointing techniques can be deployed to divide an application into small sections. At a checkpoint, the important system information and process states are saved to stable storage and a fault-detection routine (comparison of states of the two processing units) is executed simultaneously. If a fault is detected, only the faulty section of an application is executed by rolling back execution to the previous correct state [46, 49].

However, checkpoints incur time as well as energy overheads. Based on different checkpointing techniques used [71], the overheads of taking checkpoints vary greatly. For simplicity, we assume that the amount of work within each checkpoint is fixed and that taking a checkpoint consumes r time units at the maximum frequency f_{max} . For ease of presentation, we define $\gamma = \frac{r}{L}$, where L is the WCET of an application.

Checkpoints can be uniformly or non-uniformly distributed within an application [59]. In this work, we consider only uniformly distributed checkpoints. Although more checkpoints result in smaller recovery sections, more checkpoint overhead will be incurred. The optimal number of checkpoints to minimize the response time [49] as well as to minimize the energy consumption has been studied [22, 59].

In what follows, we first explore the applicability of checkpointing with respect to system load, number of recovery sections and checkpoint overhead. Then, we study the optimal number of checkpoints to minimize the expected energy consumption with a given performability goal or to maximize the performability with limited energy budget.

Suppose that there are n checkpoints within an application. When the application and all recovery sections are executed at the maximum frequency f_{max} , the maximum number of recovery sections that can be scheduled within the application's deadline is:

$$b_{max} = \left\lfloor \frac{D}{L/n + r} \right\rfloor - n = \left\lfloor \frac{D}{L/n + r} - n \right\rfloor \quad (5.11)$$

Here, we assume that the recovery overhead used to restore the previous correct state is the same as taking one checkpoint. Differentiating Equation 5.11 with respect to n , we can find that, when the number of checkpoints is $n = \left\lfloor \frac{\sqrt{DL-L}}{r} \right\rfloor$ or $n = \left\lceil \frac{\sqrt{DL-L}}{r} \right\rceil$, the maximum number of recovery sections that can be scheduled within the application's deadline is (recall that system load is defined as $\sigma = \frac{L}{D}$):

$$b_{max} = \left\lfloor \frac{(\sqrt{D} - \sqrt{L})^2}{r} \right\rfloor = \left\lfloor \frac{(1 - \sqrt{\sigma})^2}{\gamma\sigma} \right\rfloor \quad (5.12)$$

When fewer or more checkpoints are used, only fewer recovery sections can be scheduled within the application's deadline because of the increase in the recovery size or the increase in overhead for checkpointing, respectively. For example, as illustrated in Figure 5.2, when $L = 12, D = 35, r = 3$, the maximum number of recovery sections is $\left\lfloor \frac{(\sqrt{35} - \sqrt{12})^2}{3} \right\rfloor = 2$ when the optimal number of checkpoints is $\left\lceil \frac{\sqrt{35 \cdot 12} - 12}{3} \right\rceil = 3$ (shown in Figure 5.2a). When 2 or 4 checkpoints are deployed, only one recovery section can be scheduled within the application's deadline (as shown in Figure 5.2b and 5.2c, respectively).

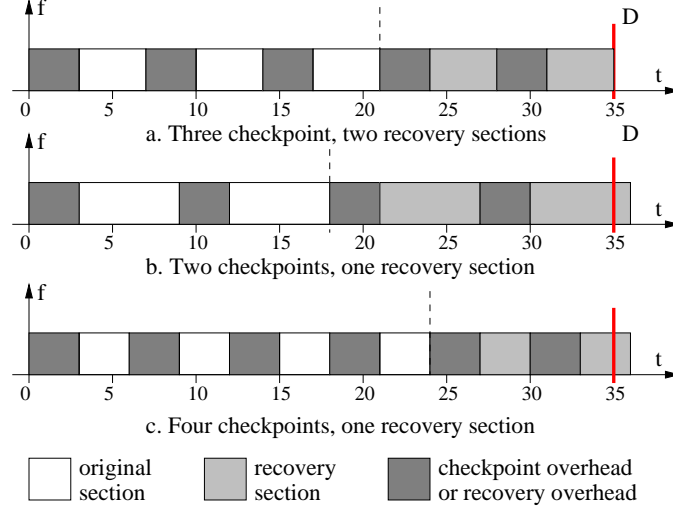


Figure 5.2: Optimal number of checkpoints for maximizing the number of recovery sections; $L = 12, D = 35, r = 3$.

Suppose that the number of recovery sections needed to achieve a give level of performance is b ($\leq b_{max}$). For example, only b faults need to be tolerated in a duplex system. With n checkpoints, we have:

$$L + nr + b\left(r + \frac{L}{n}\right) \leq D \quad (5.13)$$

which can be rewritten as

$$\gamma\sigma n^2 - (1 - \sigma - b\gamma\sigma)n + b\sigma \leq 0 \quad (5.14)$$

From Equation 5.14, in order to have a real (non-imaginary) solution for n , we should have:

$$(1 - \sigma - b\gamma\sigma)^2 - 4b\gamma\sigma^2 \geq 0 \quad (5.15)$$

Let σ_{up} be the upper bound on the system load that can be handled with b recovery sections and checkpointing overhead γ . From Equation 5.15, we can derive σ_{up} :

$$\sigma_{up} \leq \frac{1}{1 + b\gamma + 2\sqrt{b\gamma}} \quad (5.16)$$

In other words, when the system load $\sigma > \sigma_{up}$, it is not feasible to schedule b recovery sections with checkpointing overhead as γ . In what follows, we assume that $\sigma \leq \sigma_{up}$.

5.1.3 Optimal Number of Checkpoints for Maximizing Performability

When there is enough energy, both processing units in a duplex system could run at the maximum frequency f_{max} with the maximum power level of $P_{max} = P_s + P_{ind} + C_{ef}f_{max}^m$. From Equation 5.12, the maximum number of recovery sections that can be scheduled within an application's deadline is b_{max} .

With a limited energy budget, we need to reduce the power level of the duplex system. That is, we need to reduce the frequency of the processing units. Notice that, from the power model discussed in Section 3.2, it is most energy efficient to scale down the processing frequency *uniformly* for both processing units within an application's deadline. Suppose that the energy budget is E_{budget} . With the deadline of an application being D , the maximum power level that a duplex system can consume is:

$$P_{budget} = \frac{E_{budget}}{D} \quad (5.17)$$

Notice that, for a duplex system, the minimum power level is consumed when both processing units run at the minimum energy efficient frequency f_{ee} . Thus, the minimum power level consumed by a duplex system is:

$$P_{min} = 2(P_s + P_{ind} + C_{ef}f_{ee}^m) = 2(\alpha + \beta + \kappa^m)P_d^{max} \quad (5.18)$$

If $P_{min} > P_{budget}$, the energy budget E_{budget} is not enough to keep a duplex system in active working state for all the time within the application's deadline. Although it is possible to put a duplex system to sleep for some time and use the energy budget to keep a duplex system active during part of the duration, for simplicity, we consider that the energy budget is not enough for a duplex system when $P_{min} > P_{budget}$.

Suppose that $P_{min} < P_{budget}$, which means that the processing units in a duplex system could run at a higher frequency than f_{ee} . Assume that the frequency is f_{budget} . There is,

$$f_{budget} = \sqrt[m]{\frac{P_{budget}}{2 \cdot P_d^{max}} - \alpha - \beta} \quad (5.19)$$

Similar to Equation 5.11, when there are n checkpoints within an application, the number of recovery sections that can be scheduled within the application's deadline at the reduced processing frequency f_{budget} is:

$$b_{budget} = \left\lfloor \frac{Df_{budget}}{L/n + r} \right\rfloor - n = \left\lfloor \frac{Df_{budget}}{L/n + r} - n \right\rfloor \quad (5.20)$$

Notice that $b_{budget} \geq 0$. From Equation 5.20, the number of checkpoints n should satisfy:

$$1 \leq n \leq \left\lfloor \frac{Df_{budget} - L}{r} \right\rfloor = \left\lfloor \frac{f_{budget} - \sigma}{\gamma\sigma} \right\rfloor \quad (5.21)$$

With n checkpoints, the length of each application section at frequency f should be $\frac{L/n+r}{f_{budget}}$, which includes the overhead of one checkpoint. However, due to the integer limitation of b_{budget} introduced by the floor operation in Equation 5.20, the amount of time allocated to each section could be:

$$t_{n,b_{budget}} = \frac{D}{n + b_{budget}} \quad (5.22)$$

Therefore, for energy efficiency, the application and its recovery sections are executed at a frequency lower than f_{budget} with each section being allocated $t_{n,b_{budget}}$ time units.

Considering Poisson distribution of faults with average fault arrival rate λ , the probability of one section having fault(s) on a duplex system is:

$$\rho_{n,b_{budget}} = 1 - e^{-2\lambda t_{n,b_{budget}}} \quad (5.23)$$

For simplicity, we assume that the average fault arrival rate λ remains the same for different processing frequencies. The case of λ depending on processing frequency and supply voltage will be further discussed in Section 5.4.2.

Define $R(x, n)$ as the probability of completing correctly n sections when up to x sections are executed. Note that it is possible to get n correct sections before all x sections are executed. $R(x, n)$ can be computed as:

$$R(x, n) = (1 - \rho_{n,b_{budget}}) \cdot R(x - 1, n - 1) + \rho_{n,b_{budget}} \cdot R(x - 1, n) \quad (5.24)$$

where the first part stands for the probability of the first section finishing correctly and the remaining sections having $n - 1$ sections being executed correctly; the second part is the

probability of the first section having fault(s) and the remaining sections having n sections being executed correctly. Equation 5.24 has the following termination conditions:

$$\begin{cases} R(i, 0) = 1 & i \geq 0 \\ R(i, i + j) = 0 & i \geq 0; j > 0 \\ R(i, i) = (1 - \rho_{n, b_{budget}})^i & i > 0 \end{cases} \quad (5.25)$$

The first condition represents the case where all the required sections have been executed correctly and no further execution is needed. The second condition represents the case where more sections than what is available are required to be executed correctly, which is impossible and has the probability of 0. The last case is where all sections need to be executed correctly.

Therefore, the performability of an application with n checkpoints and b_{budget} recovery sections, which is the probability of having n sections being executed correctly among all $n + b_{budget}$ sections, can be expressed as:

$$R_{n, b_{budget}} = R(n + b_{budget}, n) \quad (5.26)$$

From Equations 5.20, 5.22, 5.23 and 5.26, it is difficult to get a close formula for the optimal number of checkpoints that maximizes the performability. However, from Equation 5.21, searching through all possible values of n , we can iteratively find the optimal number of checkpoints that maximizes the performability.

Alternatively, with the intuition that more recovery sections lead to higher levels of performability, we may approximate the solution by finding the number of checkpoints that results in the maximum number of recovery sections. Differentiating Equation 5.20 with respect to n , we find that, when the number of checkpoints is $n = \left\lfloor \frac{\sqrt{Df_{budget}L-L}}{r} \right\rfloor$ or $n = \left\lceil \frac{\sqrt{Df_{budget}L-L}}{r} \right\rceil$, the maximum number of recovery sections is:

$$b_{budget, max} = \left\lfloor \frac{(\sqrt{Df_{budget}L-L} - \sqrt{L})^2}{r} \right\rfloor \quad (5.27)$$

As an example, assume that checkpoint overhead is $r = 4$, for an application with $L = 30$ and $D = 100$, Figure 5.3 shows the probability of failure (i.e., $1 - \text{performability}$) and

the corresponding number of recovery sections for different numbers of checkpoints. Here, two cases are considered. When there is no energy limitation, the application is executed at the maximum frequency $f_{max} = 1$ and the corresponding maximum supply voltage. For the case of limited energy budget, suppose that the application is executed at frequency $f = 0.8$ and the corresponding voltage level. For illustration, different fault rates, $\lambda = 10^{-4}$ and $\lambda = 10^{-3}$, are assumed for the application being executed at $f_{max} = 1$ and $f = 0.8$, respectively.

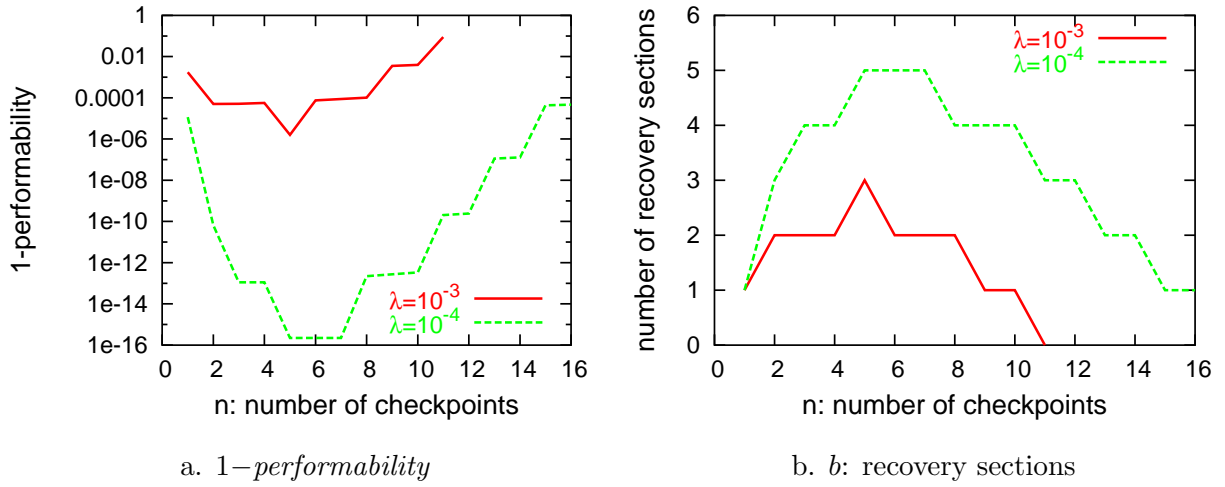


Figure 5.3: Performability and number of recovery sections.

From the figure, we can see that the optimal number of checkpoints that leads the highest performability results in the maximum number of recovery sections. However, due to the rounding effects, different numbers of checkpoints may result in the same number of recovery sections, which lead to slight difference on performability. For example, for the case of $f = 1.0$ ($\lambda = 10^{-4}$), the maximum number of recovery sections (5) can be obtained when the number of checkpoints is 5, 6 or 7, which leads to almost the same performability.

Thus, instead of searching through all possible numbers of checkpoints to get the optimal number of checkpoints that leads to the highest performability, we may find the number of checkpoints that results in the maximum number of recovery sections in constant time. This leads to the highest (or close to the highest) performability.

5.1.4 Optimal Number of Checkpoints for Energy Minimization

In this section, we focus on finding the optimal number of checkpoints that minimizes energy consumption for executing an application while achieving a given level of performability. If an application needs b ($b \leq b_{max}$) recovery sections (e.g., to tolerate up to b faults a duplex system), from Equation 5.14, the number of checkpoints n should satisfy:

$$\left\lceil \frac{(1 - \sigma - b\gamma\sigma) - \sqrt{\Delta}}{2\gamma\sigma} \right\rceil \leq n \leq \left\lfloor \frac{(1 - \sigma - b\gamma\sigma) + \sqrt{\Delta}}{2\gamma\sigma} \right\rfloor \quad (5.28)$$

where $\Delta = (1 - \sigma - b\gamma\sigma)^2 - 4b\gamma\sigma^2$.

5.1.4.1 Expected Energy Consumption For an application with b recovery sections and n checkpoints, where n satisfies Equation 5.28, the amount of slack is $D - (n + b)(\frac{L}{n} + r)$. As discussed in Section 5.1.1, considering b_e -pessimism ($b_e \leq b$), the slack can be used to scale down the processing frequency when executing the original n sections and the first b_e recovery sections. For convenience, we use *expected sections* to refer to the original n sections and the first b_e recovery sections. The remaining recovery sections are executed at the maximum frequency f_{max} if needed. The time allocated to each of expected sections and the corresponding frequency are (recall that the minimum energy efficient frequency is f_{ee}):

$$t_{expected} = \min \left\{ \frac{D - (b - b_e)(\frac{L}{n} + r)}{n + b_e}, \frac{L/n + r}{f_{ee}} \right\} \quad (5.29)$$

$$f_{expected} = \max \left\{ \frac{(n + b_e)(\frac{L}{n} + r)}{D - (b - b_e)(\frac{L}{n} + r)}, f_{ee} \right\} \quad (5.30)$$

Thus, the energy consumption for the execution of one expected section on a duplex system is:

$$E_{expected} = 2(P_s + P_{ind} + C_{ef}f_{expected}^m)t_{expected} \quad (5.31)$$

With Poisson distribution of faults with average fault arrival rate λ , the probability of one expected section having faults on a duplex system is:

$$\rho_{expected} = 1 - e^{-2\lambda t_{expected}} \quad (5.32)$$

Similarly, we can get the energy consumption and the probability of having fault during the execution in any remaining recovery sections as:

$$E_{rem} = 2(P_s + P_{ind} + C_{ef}f_{max}^m)L \quad (5.33)$$

$$\rho_{rem} = 1 - e^{-2\lambda L} \quad (5.34)$$

Notice that the original n sections are always executed. To find out the probability of recovery sections being executed, considering that expected sections and remaining recovery sections have different probabilities of failure, we extend the definition of $R(x, n)$ as follows:

$$R(x, n) = (1 - \rho_1) \cdot R(x - 1, n - 1) + \rho_1 \cdot R(x - 1, n) \quad (5.35)$$

where ρ_1 is the probability of the first section being not executed correctly. And the extended termination conditions are:

$$\begin{cases} R(i, 0) = 1 & i \geq 0 \\ R(i, i + j) = 0 & i \geq 0; j > 0 \\ R(i, i) = \prod_{j=1}^i (1 - \rho_j) & i > 0 \end{cases} \quad (5.36)$$

where ρ_j is the probability of having faults during the execution of the j^{th} section and

$$\rho_i = \begin{cases} \rho_{expected} = 1 - e^{-2\lambda t_{expected}} & i \leq n + b_e \\ \rho_{rem} = 1 - e^{-2\lambda L} & i > n + b_e \end{cases} \quad (5.37)$$

Therefore, the probability of the i^{th} recovery section being executed is the probability of fewer than n sections being correctly executed within the execution of $n + i - 1$ sections, which is $1 - R(n + i - 1, n)$. Thus, the *expected energy consumption* is:

$$E_{exp} = E_{expected} \left(n + \sum_{i=1}^{b_e} (1 - R(n + i - 1, n)) \right) + E_{rem} \sum_{i=b_e+1}^b (1 - R(n + i - 1, n)) \quad (5.38)$$

From Equation 5.38, it is difficult to get a close formula for the optimal number of checkpoints that minimizes the expected energy consumption E_{exp} . However, from Equation 5.28, searching through all possible values of n , we can iteratively find the optimal number of checkpoints that minimizes the expected energy consumption.

5.1.4.2 Fault-Free Energy Consumption Considering that faults are rare, we could be optimistic (i.e., $b_e = 0$) and use all the slack to scale down the processing frequency of the original n sections. In this section, we focus on finding the optimal number of checkpoints that minimizes the *fault-free energy consumption*. Recall that we are considering a duplex system and the processing units are always on due to the prohibitive overhead of turning a processing unit on/off. Thus, the fault-free energy consumption is:

$$E_{ff} = 2 \left(P_s D + (P_{ind} + C_{ef} f_{expected}^m) \frac{(L + nr)}{f_{expected}} \right) \quad (5.39)$$

where $f_{expected}$ is given by Equation 5.30 with $b_e = 0$. When $\frac{L+nr}{D-b(\frac{L}{n}+r)} < f_{ee}$, that is, $0 < \frac{\sigma+n\gamma\sigma}{1-b(\gamma\sigma+\frac{\sigma}{n})} \leq \kappa$, the n original sections are executed at frequency f_{ee} and the fault-free energy consumption is:

$$E_{ff} = 2 \left(\alpha + (\beta + \kappa^m) \frac{\sigma + n\gamma\sigma}{\kappa} \right) P_d^{max} D \quad (5.40)$$

Noting that $\frac{\partial E_{ff}}{\partial n} > 0$, we conclude that the fault-free energy consumption E_{ff} is minimized at the smallest n that satisfies $0 < \frac{\sigma+n\gamma\sigma}{1-b(\gamma\sigma+\frac{\sigma}{n})} \leq \kappa$. For any $n(\geq 1)$, we have $0 < \frac{\sigma+n\gamma\sigma}{1-b(\gamma\sigma+\frac{\sigma}{n})}$. From $\frac{\sigma+n\gamma\sigma}{1-b(\gamma\sigma+\frac{\sigma}{n})} \leq \kappa$, we can get a quadratic equation in n :

$$\gamma\sigma n^2 - (\kappa - \sigma - b\kappa\gamma\sigma)n + b\kappa\sigma \leq 0 \quad (5.41)$$

Solving this equation, we get:

$$\frac{(\kappa - \sigma - b\kappa\gamma\sigma) - \sqrt{\Delta}}{2\gamma\sigma} \leq n \leq \frac{(\kappa - \sigma - b\kappa\gamma\sigma) + \sqrt{\Delta}}{2\gamma\sigma} \quad (5.42)$$

where $\Delta = (\kappa - \sigma - b\kappa\gamma\sigma)^2 - 4b\gamma\sigma^2\kappa$. From the above equation, to have a real (non-imaginary) n , we must have $\Delta \geq 0$. That is, $\sigma \leq \frac{\kappa}{1+b\kappa\gamma+2\sqrt{b\kappa\gamma}} \stackrel{def}{=} \sigma_{\kappa,\gamma}$. In other words, in order to execute the original n sections of an application at frequency f_{ee} , the system load should be smaller than $\sigma_{\kappa,\gamma}$. Thus, the optimal number of checkpoints to minimize E_{ff} in this case is:

$$n = \max \left\{ 1, \frac{(\kappa - \sigma - b\kappa\gamma\sigma) - \sqrt{\Delta}}{2\gamma\sigma} \right\} \quad (5.43)$$

When system load is higher ($\sigma_{\kappa,\gamma} < \sigma \leq \sigma_{up}$), where σ_{up} is the upper bound on system load that can be handled with checkpoint overhead r and b recovery sections (see Section 5.1.2), we will have $\kappa < \frac{\sigma+n\gamma\sigma}{1-b(\gamma\sigma+\frac{\sigma}{n})}$. Therefore the original n sections of an application are executed at frequency $f_{expected} = \frac{L+nr}{D-b(\frac{L}{n}+r)}$ and the fault-free energy consumption is:

$$E_{ff} = 2 \left[\alpha + \left(\beta + \left(\frac{\sigma + n\gamma\sigma}{1 - b(\gamma\sigma + \frac{\sigma}{n})} \right)^m \right) (1 - b(\gamma\sigma + \frac{\sigma}{n})) \right] P_d^{max} D \quad (5.44)$$

Notice that any n will satisfy $\kappa < \frac{\sigma+n\gamma\sigma}{1-b(\gamma\sigma+\frac{\sigma}{n})}$ when $\sigma_{\kappa,\gamma} < \sigma \leq \sigma_{up}$. Differentiating Equation 5.44 and setting $\frac{\partial E_{ff}}{\partial n} = 0$, we find that the fault-free energy consumption E_{ff} is minimized when n satisfies the following equation:

$$mn^2\gamma(1 - b\gamma\sigma - \frac{b\sigma}{n})(\sigma + n\gamma\sigma)^{m-1} + \beta(1 - b\gamma\sigma - \frac{b\sigma}{n})^m - (m-1)(\sigma + n\gamma\sigma)^m = 0 \quad (5.45)$$

We could not find a close form for the solution n . However, for given b, m, β, γ and σ , the value of n that satisfies the above equation can be found iteratively.

In summary, when the system load is low (i.e., $0 \leq \sigma \leq \frac{\kappa}{1+b\kappa\gamma+2\sqrt{b\kappa\gamma}} \stackrel{def}{=} \sigma_{\kappa,\gamma}$), the optimal number of checkpoints n to minimize the fault-free energy consumption E_{ff} is given by Equation (5.43); when the system load is high (i.e., $\sigma_{\kappa,\gamma} < \sigma \leq \frac{1}{1+b\gamma+2\sqrt{b\gamma}} \stackrel{def}{=} \sigma_{up}$), the optimal number of checkpoints n to minimize E_{ff} can be solved iteratively. If the system load is very high (i.e., $\sigma > \sigma_{up}$), checkpointing is not applicable.

5.1.5 Evaluations of Roll-Back Recovery with Checkpoints

When an application is executed on a higher levels of modular redundant system (e.g., triple modular redundant, TMR, systems), faults can be tolerated/masked through majority voting [69]. However, by deploying more processing units, higher levels of modular redundancy imply more energy consumption.

For comparison, we consider a TMR system and compare the energy consumption and system performability for Duplex with checkpoints and TMR to tolerate one fault. Notice that, to tolerate a single fault, TMR does not need recovery and all slack can be used to scale down the frequency of the processing units for energy savings. We vary a number

of parameters: system power characteristics (sleep power P_s and frequency-independent power P_{ind}), checkpoint overhead (r) and system load (σ) to see how they affect the energy consumption and performability of Duplex and TMR.

As in Chapter 4, $m = 3$ and $P_d^{max} = 1$ are used in our analysis. Recall that $P_s = \alpha P_d^{max} = \alpha$ and $P_{ind} = \beta P_d^{max} = \beta$.

5.1.5.1 Optimal Number of Checkpoints The optimal number of checkpoints for Duplex to minimize energy consumption is determined by the frequency-independent power β , the checkpointing overhead γ and the system load σ . Figure 5.4 shows the optimal number of checkpoints for a duplex system with different frequency-independent active power and different checkpoint overhead ($\gamma = 0.01, 0.05, 0.1$ corresponding to Dup-0.01, Dup-0.05 and Dup-0.1, respectively) under different system loads.

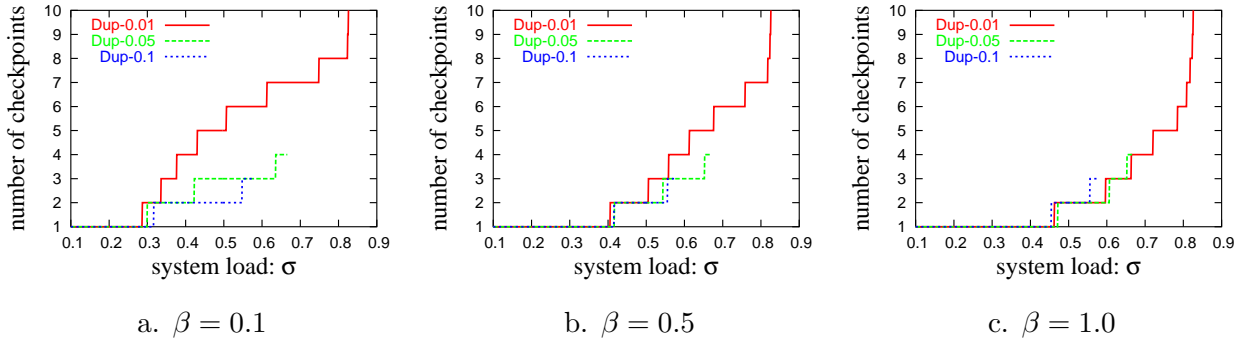


Figure 5.4: Optimal number of checkpoints for Duplex; $\alpha = 0.1$.

From the figure, we can see that Duplex is only applicable when the system load is low and/or the overhead of checkpoints is small. With checkpoint overhead increasing, the maximum system load Duplex can handle decreases. As expected, the optimal number of checkpoints increases when the checkpoint overhead decreases since more checkpoints can be used with smaller checkpoint overhead. The optimal number of checkpoints decreases when frequency-independent active power (β) increases. The reason is that the minimum energy efficient frequency f_{ee} is higher with higher frequency-independent active power. Thus, the application is able to run at f_{ee} with fewer number of checkpoints for the same system load.

5.1.5.2 Energy Efficient Regions To tolerate a single fault, Duplex needs one recovery while TMR does not. Therefore, TMR is feasible when $\sigma \leq 1$. However, for Duplex, it is feasible only when $\sigma \leq \sigma_{up} = \frac{1}{1+\gamma+2\sqrt{\gamma}}$ (see Equation 5.16) for a given checkpoint overhead $r = \gamma L$.

When Duplex is not feasible, we can only use TMR. When both Duplex and TMR are feasible, the minimum fault-free energy consumption for Duplex, E_{dup} , can be obtained by deploying the optimal number of checkpoints as discussed in Section 5.1.4.2. The minimum fault-free energy consumption for TMR, E_{tmr} , can be obtained when all three processing units run at the frequency of $f_{tmr} = \max\{\sigma f_{max}, f_{ee}\}$. When E_{dup} is less than E_{tmr} , Duplex should be used; otherwise, use TMR.

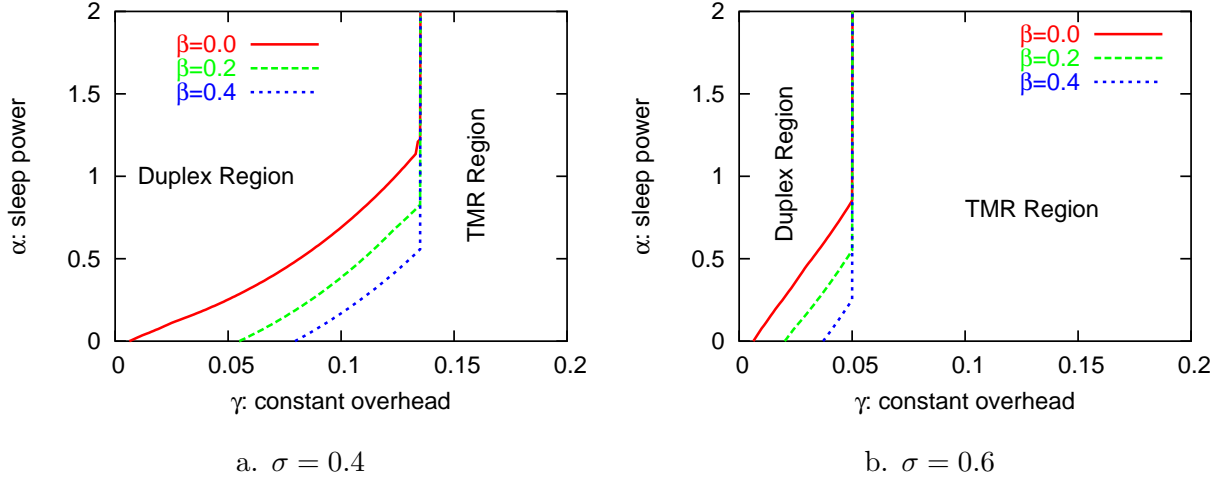


Figure 5.5: Energy efficient regions for Duplex and TMR

Suppose the application's deadline is $D = 1000$ time units. We consider two different loads $\sigma = 0.4$ ($L = 400$) and $\sigma = 0.6$ ($L = 600$). Figure 5.5 shows the energy efficient regions for Duplex and TMR with different checkpoint overheads and system power characteristics. From the figures, we can see that when the system load is higher, the maximum checkpoint overhead that Duplex can use is smaller. Furthermore, lower sleep and/or frequency-independent active power favors TMR even when Duplex is applicable. These observations are the same as reported in [22].

5.1.5.3 System Performability With the assumption that the inter-arrival time of faults follows Poisson distribution and the average failure rate is λ , the probability of failure on one processing unit during the period of D is $\rho(D) = 1 - e^{-\lambda D}$. Since the deadline D is an application specific parameter, in the following discussion we assume that $\rho(D) = 10^{-3}, 10^{-4}$ and 10^{-5} . Figure 5.6 shows the probability of failure ($1 - \text{performability}$) for Duplex and TMR with different values of $\rho(D)$ (i.e., different failure rates). For Duplex, the optimal number of checkpoints that minimizes energy consumption is used. Notice that lower probability of failure means higher performability.

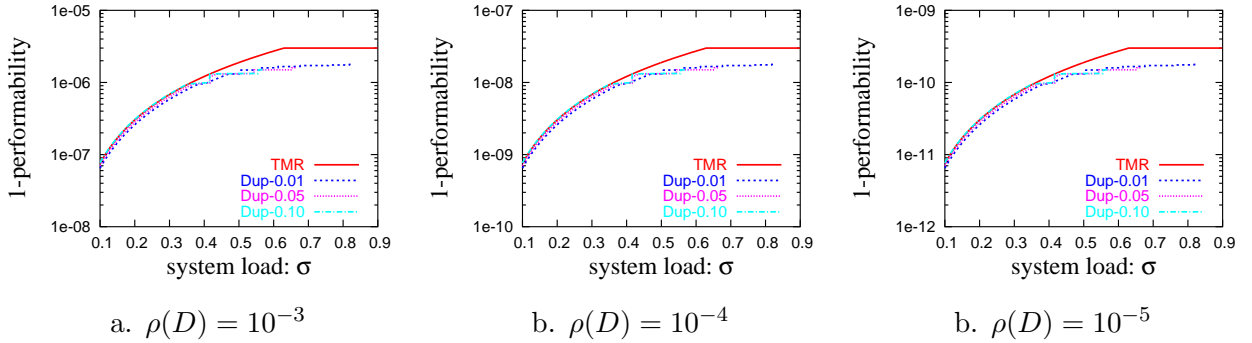


Figure 5.6: The probability of failure ($1 - \text{performability}$) for Duplex and TMR.

In Figure 5.6, we assume that $\alpha = 0.1$ and $\beta = 0.5$. As expected, when the load σ increases, the performabilities decrease since both Duplex and TMR use more time to execute the application. The reason for the decreased performability is that, with the inter-arrival time of faults following a Poisson distribution, the longer a processing unit runs, the higher the probability it fails and the lower the performability is. Also note that different checkpoint overheads have no significant effect on the performability achieved by Duplex. With one recovery section, Duplex achieves comparable levels of performability as that of TMR, especially with low loads where all executions are performed at the minimum energy efficient frequency.

5.2 OPTIMISTIC MODULAR REDUNDANCY

From the discussion in the last section, compared with Duplex that deploys checkpoints, only TMR is applicable when the overhead of checkpoints is large and/or the system load is high. However, TMR uses one more processing unit and consumes more energy, especially for larger sleep powers and frequency-independent powers. In [22], an *optimistic triple modular redundancy (OTMR)* scheme was proposed to reduce the energy consumption in a traditional TMR system. Expecting that faults are rare, OTMR turns off or scales down one processing unit provided that it can catch up and finish the computation before the application's deadline if the other two processing units do encounter faults.

Considering the general system power model, we first explore the optimal frequency setting for OTMR to minimize system energy consumption (Section 5.2.1). Then, for that optimal frequency setting, assuming a Poisson distribution of faults, the performability of OTMR is derived (Section 5.2.2). The detailed comparison between OTMR and TMR on both energy consumption and performability is presented (Section 5.2.3) and the idea is extended to an *optimistic N-modular redundancy (ONMR)* scheme (Section 5.2.4).

5.2.1 Optimal Frequency Setting for OTMR

For the OTMR scheme, the processing frequency for the first two processing units is *crucial* in determining the amount of slack reserved for the third processing unit, *when* the third processing unit should begin to run and at *what* frequency. Clearly, the optimal frequency setting for OTMR to minimize energy consumption depends on system loads (i.e., the amount of available slack). Recall that f_{ee} is the minimum energy efficient frequency (Section 3.2). If the system load is very low, we can reserve enough slack for the third processing unit and run the first two processing units at f_{ee} as shown in Figure 5.7a.

In the figure, the white rectangle represents the execution of the application on the first two processing units and the dark shadowed rectangles are the reserved times for the third processing unit. The width of a rectangle represents processing frequency and the height represents execution time. When the system load is higher, based on the amount of

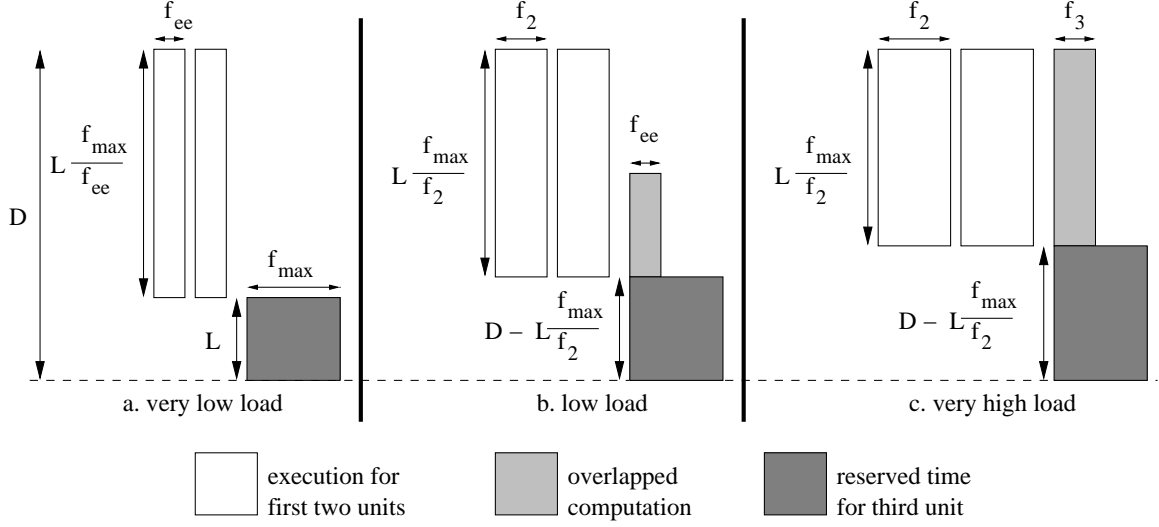


Figure 5.7: Optimal frequency setting for OTMR under different system loads: a. very low system load ($0 \leq \sigma \leq \frac{\kappa}{1+\kappa}$); b. low system load ($\frac{\kappa}{1+\kappa} < \sigma \leq \kappa$); c. very high system load ($\frac{1}{2-\kappa} < \sigma \leq 1$).

reserved slack for the third processing unit and its processing frequency, the third processing unit may need to execute part of the application concurrently with the first two processing units (defined as *overlapped computation*) as shown Figures 5.7bc, where the light shadowed rectangles are the overlapped computations in the third processing unit.

5.2.1.1 Expected Energy Consumption Suppose that the optimal frequency for the first two processing units is f_2 and f_{3r} is the optimal frequency for the third processing units to execute the application during the reserved time. The optimal frequency f_3 that the third processing unit uses to perform the overlapped computation will depend on f_2 and f_{3r} . Recall that normalized frequency is used and $f_{max} = 1$. With the first two processing units running at f_2 , the amount of reserved time for the third processing unit is $D - \frac{L}{f_2}$. Thus, the amount of overlapped computation is $L - f_{3r}(D - \frac{L}{f_2})$. Thus,

$$f_3 = \frac{L - f_{3r}(D - \frac{L}{f_2})}{\frac{L}{f_2}} = \frac{(L - f_{3r}(D - \frac{L}{f_2}))f_2}{L} \quad (5.46)$$

Recall that the minimum energy efficient frequency is f_{ee} and the maximum frequency is f_{max} . Therefore:

$$f_{ee} \leq f_2, f_3, f_{3r} \leq f_{max} = 1 \quad (5.47)$$

The power consumption in a processing unit at frequency f is modeled as $P(f) = P_s + \hbar(P_{ind} + C_{ef}f^m)$ (see Section 3.2). Since the processing units are assumed to be always on, the sleep power P_s is always consumed on all processing units within the time period considered. We use $P_a(f)$ to stand for active power at frequency f and $P_a(f) = P_{ind} + C_{ef}f^m$. Considering the probabilities of performing computations, the *expected energy consumption* of an OTMR system is:

$$E_{exp} = 3P_s D + 2 \cdot P_a(f_2) \cdot t_2 + P_a(f_3) \cdot t_3 + (1 - R_2)P_a(f_{3r}) \cdot t_{3r} \quad (5.48)$$

where

$$t_2 = \frac{L}{f_2} \quad (5.49)$$

$$t_{3r} = \max \left\{ \frac{L}{f_{ee}}, (D - t_2) \right\} \quad (5.50)$$

$$t_3 = \frac{L - t_{3r}f_{3r}}{f_3} \quad (5.51)$$

Here, t_2 , t_3 and t_{3r} are the execution times for the computation on the first two processing units, for the overlapped computation on the third processing unit and for the computation during reserved time on the third processing unit, respectively.

The first part in Equation 5.48 is the energy consumption in a system due to sleep power and is always consumed. The second and the third parts are active energy consumed by the first two processing units and the third processing unit, respectively. They are also always consumed whether the execution on the first two processing units has errors or not. The last part is the active energy consumed by the third processing unit during the reserved time. This has a probability $(1 - R_2)$ of being executed where R_2 is the probability of having no fault(s) during the execution on the first two processing units. Assuming that faults follow a Poisson distribution with an average fault arrival rate λ , we have:

$$R_2 = (1 - (1 - e^{-\lambda t_2}))^2 = e^{2(-\lambda t_2)} \quad (5.52)$$

For given L, D and λ , from Equations 5.46, 5.48, 5.49, 5.50, 5.51 and 5.52, we can see that E_{exp} is a function of f_2 and f_{3r} . However, we cannot get close formula solutions of f_2 and f_{3r} to minimize E_{exp} due to the exponential component in R_2 . Notice that $f_{ee} \leq f_2, f_{3r} \leq f_{max}$. Searching over the feasible values of f_2 and f_{3r} , we may approximate the solutions iteratively, especially for systems that have only a few frequency levels.

5.2.1.2 Fault Free Energy Consumption To simplify the problem, we assume that the third processing unit runs at the maximum frequency during the reserved time (i.e., $f_{3r} = f_{max}$) and focus on minimizing the *fault-free energy consumption* that is given as:

$$E_{ff} = 3P_s D + 2 \cdot P_a(f_2) \cdot t_2 + P_a(f_3) \cdot t_3 \quad (5.53)$$

Without loss of generality, suppose that the optimal frequency for the first two processing units is $f_2 = x f_{max}$. The reserved time for the third processing unit is $D - \frac{L}{x}$. If $D - \frac{L}{x} \geq L$ (i.e., the reserved time is enough for the third processing unit to finish the computation at the maximum frequency f_{max}), no computation needs to be done concurrently and the third processing unit just sleeps initially (see Case 1 below). Otherwise, the amount of overlapped computation is $L - (D - \frac{L}{x})$ at frequency $f_3 = \max\{f_{ee}, \frac{L - (D - \frac{L}{x})}{\frac{L}{x}} f_{max}\} = \max\{f_{ee}, (1 - \frac{1-\sigma}{\sigma} x) f_{max}\}$. Therefore, due to the minimum energy efficient frequency f_{ee} , in order to find the optimal frequencies, we need to consider the following cases corresponding to different system loads. Notice that $f_2 \geq \max\{\sigma f_{max}, f_{ee}\}$. That is, $x \geq \max\{\sigma, \kappa\}$.

Case 1: $0 < \sigma \leq \frac{\kappa}{1+\kappa}$ (system load is very low).

In this case, we have $L \leq D - \frac{L}{\kappa}$ (from $\sigma \leq \frac{\kappa}{1+\kappa}$). That is, there is enough reserved time and the third unit just sleeps initially. The optimal frequency for the first two processing units is f_{ee} as shown in Figure 5.7a. The minimum fault-free energy consumption for OTMR is:

$$\begin{aligned} E_{OTMR} &= 3P_s D + 2(P_{ind} + C_{ef} f_{ee}^m) \frac{L}{f_{ee}} \\ &= \left(3\alpha + 2(\beta + \kappa^m) \frac{\sigma}{\kappa}\right) P_d^{max} D \end{aligned} \quad (5.54)$$

Case 2: $\frac{\kappa}{1+\kappa} < \sigma \leq \kappa$ (system load is low).

In this case, the third processing unit may have to execute part of the application concurrently with the first two processing units as shown in Figure 5.7b. However, as discussed below, the frequency at which the third processing unit executes the overlapped computation is not higher than f_{ee} .

Notice that, when the reserved time for the third processing unit is L , the first two processing units should run at frequency $\frac{L}{D-L}f_{max} = \frac{\sigma}{1-\sigma}f_{max}$, which is higher than f_{ee} (because $\frac{\kappa}{1+\kappa} < \sigma$ and $f_{ee} = \kappa f_{max}$). Running at a frequency higher than $\frac{\sigma}{1-\sigma}f_{max}$ will needlessly increase the energy consumption for the first two processing units. Since the optimal frequency for the first two processing units $f_2 = xf_{max}$ is also limited by f_{max} , we have $f_2 \leq \min\{\frac{\sigma}{1-\sigma}f_{max}, f_{max}\}$. That is, $x \leq \min\{\frac{\sigma}{1-\sigma}, 1\}$. Thus, we have $L - (D - \frac{L}{x}) \geq 0$. Recall that the optimal frequency for the third processing unit to execute the overlapped computation is $f_3 = \max\{f_{ee}, (1 - \frac{1-\sigma}{\sigma}x)f_{max}\}$. Since $f_2 \geq f_{ee}$, that is, $x \geq \kappa \geq \sigma$, we have $(1 - \frac{1-\sigma}{\sigma}x)f_{max} \leq (1 - \frac{1-\sigma}{\sigma}\sigma)f_{max} = \sigma f_{max} \leq f_{ee}$. Thus, $f_3 = f_{ee}$. However, the start time for the third processing unit to execute the overlapped computation is determined by x and σ . Therefore, the fault-free energy consumption for OTMR is:

$$\begin{aligned} E_{OTMR} &= 3P_s D + 2(P_{ind} + C_{ef}f_{b1}^m)\frac{L}{x} + (P_{ind} + C_{ef}f_{ee}^m)\frac{L - (D - \frac{L}{x})}{\kappa} \\ &= \left(3\alpha + 2(\beta + x^m)\frac{\sigma}{x} + (\beta + \kappa^m)\frac{\sigma - (1 - \frac{\sigma}{x})}{\kappa}\right) P_d^{max} D \end{aligned} \quad (5.55)$$

Differentiating the above equation with respect to x , we get:

$$\frac{\partial E_{OTMR}}{\partial x} = \frac{2(m-1)x^m - 2\beta - (\frac{\beta}{\kappa} + \kappa^{m-1})}{x^2} \sigma = 0 \quad (5.56)$$

Solving the above equation, we conclude that E_{OTMR} is minimized when

$$x = \sqrt[m]{\frac{2\beta + (\frac{\beta}{\kappa} + \kappa^{m-1})}{2(m-1)}} \stackrel{def}{=} x_{\beta, \kappa, m} \quad (5.57)$$

subject to $\kappa \leq x \leq \min\{\frac{\sigma}{1-\sigma}, 1\}$. Thus, if $x_{\beta, \kappa, m} \leq \kappa$, E_{OTMR} is minimized when $x = \kappa$; otherwise, E_{OTMR} is minimized when $x = \min\{1, \frac{\sigma}{1-\sigma}, x_{\beta, \kappa, m}\}$.

Case 3: $\kappa < \sigma \leq \frac{1}{2-\kappa}$ (system load is high).

Because $f_2 = xf_{max}$ and $f_3 = \max\{f_{ee}, (1 - \frac{1-\sigma}{\sigma}x)f_{max}\}$, the frequency $(1 - \frac{1-\sigma}{\sigma}x)f_{max}$ is smaller than f_{ee} if $x \geq \frac{1-\kappa}{1-\sigma}$. In this case, $f_3 = f_{ee}$ (same as in Figure 5.7b) and the

optimal x to minimize E_{OTMR} can be solved as in Case 2. However, if $x < \frac{1-\kappa}{1-\sigma}\sigma$, we have $(1 - \frac{1-\sigma}{\sigma}x)f_{max} > f_{ee}$ and the third processing unit will run at frequency $f_3 = (1 - \frac{1-\sigma}{\sigma}x)f_{max}$ (as in Figure 5.7c). The optimal x to minimize E_{OTMR} can be found iteratively as will be discussed next in Case 4. As the result, the optimal x is the one that results in smaller E_{OTMR} among the two sub cases.

Case 4: $\frac{1}{2-\kappa} < \sigma \leq 1$ (the system load is very high).

From $\frac{1}{2-\kappa} < \sigma$, we have $1 < \frac{1-\kappa}{1-\sigma}\sigma$. Note that $x \leq \min\{\frac{\sigma}{1-\sigma}, 1\}$ as discussed in Case 2. Therefore, $x \leq 1 < \frac{1-\kappa}{1-\sigma}\sigma$, that is, $\kappa < 1 - \frac{1-\sigma}{\sigma}x$. Hence the third processing unit needs to execute the overlapped computation at frequency $f_3 = yf_{max} = (1 - \frac{1-\sigma}{\sigma}x)f_{max} > f_{ee}$ as shown in Figure 5.7c. Thus, the fault-free energy consumption for OTMR is:

$$\begin{aligned} E_{OTMR} &= 3P_s D + 2(P_{ind} + C_{ef}f_2^m)\frac{L}{x} + (P_{ind} + C_{ef}f_3^m)\frac{L - (D - \frac{L}{x})}{y} \\ &= \left(3\alpha + 2(\beta + x^m)\frac{\sigma}{x} + (\beta + y^m)\frac{\sigma - (1 - \frac{\sigma}{x})}{y}\right) P_d^{max} D \end{aligned} \quad (5.58)$$

Setting $\frac{\partial E_{OTMR}}{\partial x} = 0$, we conclude that E_{OTMR} is minimized when x satisfies the following equation subject to $\sigma < x \leq \min\{\frac{\sigma}{1-\sigma}, 1\}$:

$$\begin{aligned} &2(m-1)\sigma x^m y^2 - (1-\sigma)(m-1)x^2 y^m + (m-1)\frac{1-\sigma}{\sigma}(x-\sigma)xy^m \\ &+ \beta(1-\sigma)x^2 - \beta\sigma y - 2\beta\sigma y^2 - \frac{(1-\sigma)\beta}{\sigma}(x-\sigma)x - \sigma y^{m+1} = 0 \end{aligned} \quad (5.59)$$

where $y = 1 - \frac{1-\sigma}{\sigma}x$. It is not clear if there is a close form for the solution x . For a given m, β and σ , however, x can be found iteratively.

In summary, to minimize the energy consumption of OTMR, when system load is very low (i.e., $\sigma \leq \frac{\kappa}{1+\kappa}$), the optimal frequency for the first two processing units is f_{ee} and the third processing unit just sleeps initially. When system load is low (i.e., $\frac{\kappa}{1+\kappa} < \sigma \leq \kappa$), the optimal frequency for the first two processing units is f_{ee} (if $x_{\beta, \kappa, m} \leq \kappa$) or $\min\{1, \frac{\sigma}{1-\sigma}, x_{\beta, \kappa, m}\}f_{max}$, while the third processing unit runs at f_{ee} , where $x_{\beta, \kappa, m} = \sqrt[m]{\frac{a\beta + (N-a)(\frac{\beta}{\kappa} + \kappa^{m-1})}{a(m-1)}}$. When system load is high (i.e., $\kappa < \sigma \leq 1$) the optimal frequencies for the processing units need to be solved iteratively.

5.2.2 Performability of OTMR

When all processing units run at their optimal frequencies to minimize energy consumption, assuming that faults follow a Poisson distribution with an average fault arrival rate λ , we can find the performability of OTMR as follows. The probability of having fault(s) on one processing unit during a period of time t is:

$$\rho(t) = 1 - e^{-\lambda t} \quad (5.60)$$

With the optimal frequency f_2 , the first two processing units run for $t_2 = \frac{L}{f_2}$ time units and the probability of having fault(s) during the execution on one processing unit is $\rho_2 = \rho(t_2) = 1 - e^{-\lambda \frac{L}{f_2}}$. For the execution of the application on the third processing unit, it is divided into two parts. Initially, the application is executed at frequency f_3 for t_3 time units (the overlapped computation); then, it is executed at frequency f_{3r} for t_{3r} time units (the reserved time). Therefore, the third processing unit will run for $(t_3 + t_{3r})$ time units and the probability of having fault(s) on the third unit is $\rho_3 = \rho(t_3 + t_{3r}) = 1 - e^{-\lambda(t_3 + t_{3r})}$. Therefore, the performability of a OTMR system is:

$$R_{OTMR} = (1 - \rho_2)^2 + 2(1 - \rho_2)\rho_2(1 - \rho_3) \quad (5.61)$$

where the first term is the probability of having no fault in the first two processing units and the second term is the probability of having fault(s) on any one of the first two processing units but no fault on the third processing unit.

5.2.3 Comparison of OTMR and Traditional TMR

In a traditional TMR scheme, the slack is used to scale down all processing units that have the same frequency. With system load as $\sigma = \frac{L}{D}$, the optimal frequency to minimize energy consumption for a TMR system and the corresponding running time are:

$$f_{TMR} = \max\{f_{ee}, \sigma f_{max}\} \quad (5.62)$$

$$t_{TMR} = \min\left\{\frac{L}{\kappa}, D\right\} \quad (5.63)$$

where $f_{ee} = \kappa f_{max}$ is the minimum energy efficient frequency. Therefore, the energy consumption for a TMR system is:

$$E_{TMR} = 3 [P_s D + (P_{ind} + C_{ef} f_{TMR}^m) t_{TMR}] \quad (5.64)$$

Given a Poisson distribution of faults with average fault arrival rate λ , the probability of having fault(s) on one processing unit would be:

$$\rho_{TMR} = \rho(t_{TMR}) = 1 - e^{-\lambda t_{TMR}} \quad (5.65)$$

Thus, the performability of a TMR system is:

$$R_{TMR} = (1 - \rho_{TMR})^3 + 3(1 - \rho_{TMR})^2 \rho_{TMR} \quad (5.66)$$

where the first term is the probability of having no faults during execution and the second term is the probability of having fault(s) only in one processing unit.

In what follows, we provide detailed comparison between OTMR and traditional TMR on both energy consumption and performability. Recall that $m = 3$ and the maximum frequency-dependent active power $P_d^{max} = 1$. The sleep power P_s is the same for TMR and OTMR and we assume that the sleep power $P_s = 0.1$. We vary the values of the frequency-independent active power as $P_{ind} = \beta P_d^{max} = 0.1, 0.5, 1.0$. Figure 5.8 shows the optimal processing frequencies for the first two processing units in OTMR (OTMR-2), the third processing unit of OTMR (OTMR-3) and all processing units in TMR (TMR), to minimize energy consumption under different system loads.

Notice that, the minimum energy efficient frequency $f_{ee} = \sqrt[m]{\frac{\beta}{m-1}} f_{max} = \kappa f_{max}$ is determined by β and m . With $m = 3$, the larger β leads to larger f_{ee} . From Figure 5.8, we can see that the optimal frequency for TMR is $\max\{f_{ee}, \sigma f_{max}\}$. For OTMR, the optimal frequency for the first two processing units is the same as that for TMR when system load is low ($\sigma \leq \frac{\kappa}{1+\kappa}$), and begins to increase sharply when system load becomes higher. By running the first two processing units faster, OTMR reserves enough time and the third

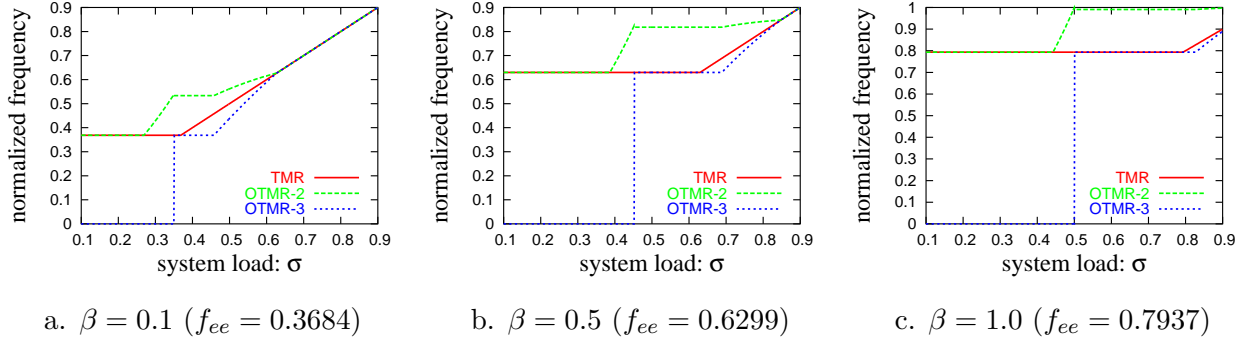


Figure 5.8: Optimal frequencies for OTMR and TMR.

processing unit could sleep more at the beginning and thus save more energy. For example, when $\beta = 0.5$ (Figure 5.8b), the optimal frequency for the first two processing units is $f_{ee} = \kappa f_{max} = 0.6299$ and the third processing unit sleeps when the load $\sigma \leq \frac{\kappa}{1+\kappa} = 0.39$; when the load is slightly higher, the optimal frequency for the first two processing units increases to $\frac{\sigma}{1-\sigma}$ and the third processing unit continues to sleep until the load reaches $\sigma = 0.4499$, at which point $\frac{\sigma}{1-\sigma} = 0.8181 = x_{\beta,\kappa,m} = \sqrt[m]{\frac{2\beta+\beta/\kappa+\kappa^{m-1}}{2(m-1)}}$. After that the first two processing units run at the optimal frequency $x_{\beta,\kappa,m} = 0.8181$ and the third processing unit begins to run at $f_{ee} = 0.6299$. However, the running time for the third processing unit is not the same as the first two processing units before load approaches $\frac{1}{2-\kappa} = 0.7299$, after which the frequency of the third processing unit is higher than $f_{ee} = 0.6299$.

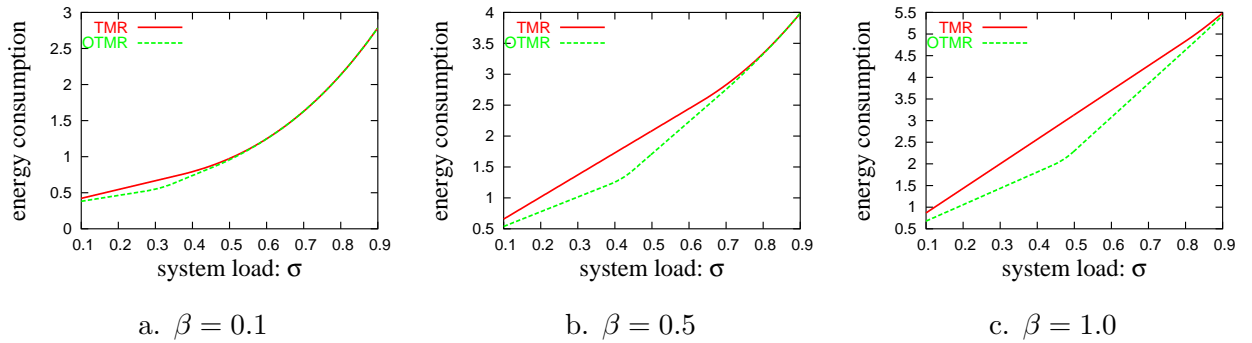


Figure 5.9: The energy consumption of OTMR and TMR.

Figure 5.9 show the minimum energy consumptions for OTMR and TMR with different system loads. Even though OTMR's first two processing units run faster than TMR, the

sleep of the third processing unit leads to OTMR consuming equal or less energy than TMR at the expense of a slight more complex frequency management scheme.

We further examine the performability achieved by OTMR and TMR when their processing frequencies are optimized for energy consumption. As in Section 5.1, we assume that $\rho(D) = 10^{-3}, 10^{-4}$ and 10^{-5} (i.e., different fault arrival rates). Figure 5.10 shows the probability of failure (i.e., $1 - \text{performability}$) for OTMR and TMR with different values of $\rho(D)$. As expected, when the load σ increases, the performabilities for all schemes decrease since all schemes use more time to execute the application. The reason is that, with the inter arrival time of faults following a Poisson distribution, the longer a processing unit runs, the higher the probability it fails and the lower the performability is. OTMR achieves slightly better performability than TMR since OTMR runs faster and uses less time to execute an application.

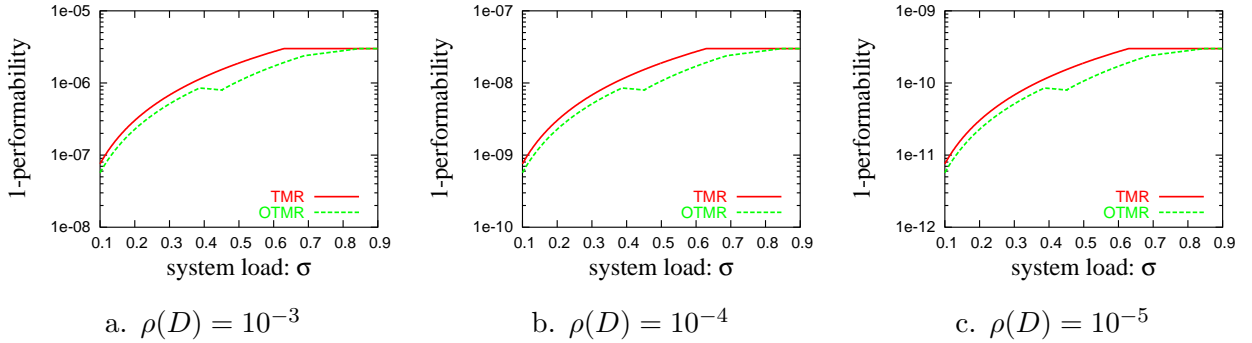


Figure 5.10: The probability of failure ($1 - \text{performability}$) of OTMR and TMR; $\beta = 0.5$.

5.2.4 Optimistic N-Modular Redundancy (ONMR)

Extending the idea of OTMR, we consider in this section a general *optimistic N-modular redundancy (ONMR)* scheme. For the *a-out-of-N* ($a < N$) modular redundancy model, ONMR puts $N - a$ processing units (defined as *backup* processing units) to sleep or scale down their processing frequency as long as they have enough reserved time to finish the execution before the application's deadline D whenever needed. The first a processing units are referred to as *primary* processing units.

As for OTMR, the optimal frequency setting for the processing units in ONNR depends on system loads. If the system load is low, we can run the primary processing units at the minimum energy efficient frequency f_{ee} while reserving enough slack for the backup processing units such that they can run *together* at f_{ee} and meet the application's deadline when there are faults during the application's execution on the primary processing units. In this case, the analysis for the optimal frequency setting for OTMR in the previous sections can be applied to ONMR directly, where the frequency for the first two processing units in OTMR corresponds to the frequency for the primary processing units in ONMR and the frequency for the third processing unit in OTMR corresponds to the frequency for the backup processing units in ONMR.

Suppose that the optimal frequency for the primary processing units is f_a , the optimal frequencies for the backup processing units are f_{b1} and f_{b2} , which correspond to the processing frequencies for the overlapped computation and the computation during the reserved time, respectively. Therefore, the primary processing units run for $t_a = \frac{L}{f_a}$ time units and, as in OTMR, for the execution of the application on the backup processing units, it is divided into two parts. Initially, the application is executed at frequency f_{b1} for t_{b1} time units (the overlapped computation); then, it is executed at frequency f_{b2} for t_{b2} time units (the reserved time). That is, the backup processing units will run for $(t_{b1} + t_{b2})$ time units.

Thus, under the optimal frequency setting for ONMR, the probability of x processing units (out of a primary processing units) having fault(s) is:

$$Pr(a, x, \lambda, t_a) = C_a^x \rho(\lambda, t_a)^x (1 - \rho(\lambda, t_a))^{a-x} \quad (5.67)$$

Similarly, the probability of y processing units (out of $N - a$ backup processing units) having fault(s) is:

$$Pr(N - a, y, \lambda, t_{b1} + t_{b2}) = C_{N-a}^y \rho(\lambda, t_{b1} + t_{b2})^y (1 - \rho(\lambda, t_{b1} + t_{b2}))^{N-a-y} \quad (5.68)$$

The performability of an ONMR system is the summation of the probability of getting x faulty processing units among the primary processing units while *at least* x processing units among the $N - a$ backup processing units get correct results. Notice that x should not be

larger than $N - a$, the number of backup processing units. That is, $0 \leq x \leq \min\{a, N - a\}$. Therefore, the performability of an ONMR system is:

$$R_{ONMR} = \sum_{x=0}^{\min\{a, N-a\}} \left[Pr(a, x, \lambda, t_a) \sum_{y=0}^{N-a-x} Pr(N-a, y, \lambda, t_{b1} + t_{b2}) \right] \quad (5.69)$$

Instead of running all the backup processing units together at the same frequency, we can apply the same idea iteratively and put the backup processing units to work at different time and different frequencies. However, the analysis of the optimal frequency setting for the processing units in ONMR will become more complex and we will explore this point in our future work.

5.3 ENERGY EFFICIENT REDUNDANCY CONFIGURATION

For single task applications, we have explored checkpointing techniques (Section 5.1) and optimistic modular redundancy (Section 5.2) for energy efficient fault tolerance. In this section, we consider a *fully parallel* application consisting of multiple independent tasks that is executed on a system with M processing units. For simplicity, we consider deterministic fault model and focus on exploring the most efficient system configuration to maximize the number of faults that can be tolerated with limited energy budget or to minimize energy consumption with a given number of faults to be tolerated.

Moreover, we assume that all the independent tasks in an application are the *same size* of one time unit. For parallel applications that consist of tasks with different sizes, checkpointing techniques may be deployed as described in Section 5.1.3 to divide tasks into sections of the same size.

Suppose that there are w independent tasks in an application. Recall that replicated execution is used to detect faults (see Section 3.3). Define a *redundant group* as the processing units that handle the same set of tasks. We assume that the application is executed on p ($p \leq \lfloor \frac{M}{2} \rfloor$) duplex groups. The application will need $n = \left\lceil \frac{w}{p} \right\rceil$ time units. Define a *section* as the execution of one task on one duplex group. If faults occur during the execution of one

task, the task becomes faulty and a *recovery section* of one time unit is needed to re-execute the faulty task.

Suppose that b time units are needed to be reserved as *backup slots*, where each backup slot has p parallel recovery sections. The schedule for executing all tasks within the application's deadline D is shown in Figure 5.11. In the figure, each white rectangle represents a section that is used to execute one task on one duplex group and the shadowed rectangles represent the recovery sections reserved for re-executing the faulty tasks. For ease of presentation, the first n time units are referred to as *primary time units* and all white rectangles are referred as *primary executions*. After scheduling the primary time units and backup slots, the amount of slack left is $D - (n + b)$, which can be used to scale down the processing frequency of the processing units and save energy.

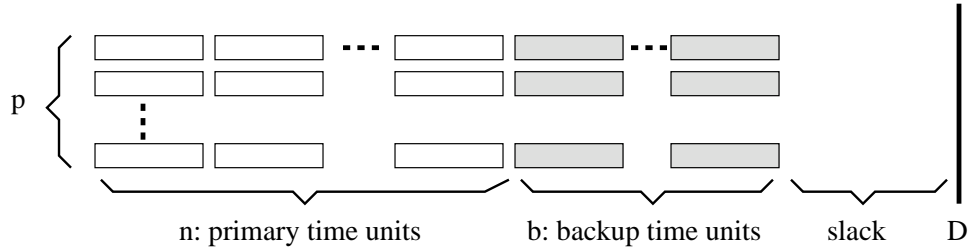


Figure 5.11: For an application consisting of $w (= n \cdot p)$ independent tasks that have WCET as one time unit, it needs n time units when it is executed on $p (\leq \lfloor \frac{M}{2} \rfloor)$ duplex groups. Suppose that b time units are reserved as backup slots and the slack time is $D - n - b$.

For a faulty task, its re-execution during a recovery section may also encounter faults. If all the recovery sections that are used to re-execute a faulty task encounter faults, then we say that there is a *recovery failure* and the faulty task needs to be re-executed again.

We first address the recovery schemes that work with parallel slack and propose an efficient adaptive parallel recovery schemes that recovers tasks in parallel (Section 5.3.1). Then, in Section 5.3.2, we discuss the combination of parallel recovery and modular redundancy. The optimal redundant configuration of the processing units to minimize system energy consumption with a fixed performability goal or to maximize system performability with limited energy budget is explored in Section 5.3.3. Finally, the analysis results are presented

in Section 5.3.4.

5.3.1 Recovery Schemes with Parallel Slack

In this section, we calculate the worst case maximum number of faults that can be tolerated during the execution of w tasks by p duplex groups with b backup slots. The addition of one more fault could cause an additional faulty task that can not be recovered and thus lead to a system failure. As a first step, we assume that the number of requests w is a multiple of p (i.e., $w = n \cdot p$, $n \geq 1$). The case of w being not a multiple of p will be discussed in Section 5.3.1.4. For different strategies of using backup slots, we consider three recovery schemes: *restricted serial recovery*, *parallel recovery* and *adaptive parallel recovery*.

Consider the example shown in Figure 5.12 where 9 tasks are executed on three duplex groups. The tasks are labeled T_1 to T_9 and there are two backup slots (i.e., six recovery sections). Again, each box in the figure represents the execution of a task on a duplex group. Suppose that tasks T_3 and T_8 become faulty on the top duplex during the third time unit and the bottom duplex during the second time unit, respectively. Task T_8 is recovered immediately during the third time unit (R_8) and the execution of task T_9 is postponed. Therefore, before using backup slots, there are two tasks to be executed/re-executed; the original task T_9 and the recovery task R_3 .

T ₁	T ₂	T ₃ ✗	R ₃		T ₁	T ₂	T ₃ ✗	R ₃	T ₉	T ₁	T ₂	T ₃ ✗	R ₃	
T ₄	T ₅	T ₆			T ₄	T ₅	T ₆	T ₉	R ₃	T ₄	T ₅	T ₆	R ₃	
T ₇	T ₈ ✗	R ₈	T ₉		T ₇	T ₈ ✗	R ₈	R ₃	T ₉	T ₇	T ₈ ✗	R ₈	T ₉	

a. Restricted serial recovery

b. Parallel recovery

c. Adaptive parallel recovery

Figure 5.12: Different recovery schemes.

5.3.1.1 Restricted Serial Recovery The restricted serial recovery scheme limits the re-execution of a faulty task to the *same* duplex. For example, Figure 5.12a shows that T_3 is recovered by R_3 on the top duplex while T_8 is recovered by R_8 on the bottom duplex.

It is easy to see that, with b backup slots, the restricted serial recovery scheme can only recover from b fault in the worst case (either during primary or backup slots). For example,

as shown in Figure 5.13a, if there is a fault that causes task T_3 to be faulty during primary execution, we can only tolerate one more fault in the worst case when the fault causes T_3 's recovery R_3 to be faulty. One additional fault could cause the second recovery RR_3 of task T_3 to be faulty and lead to system failure since the recovery of the faulty tasks is restricted to the same duplex group.

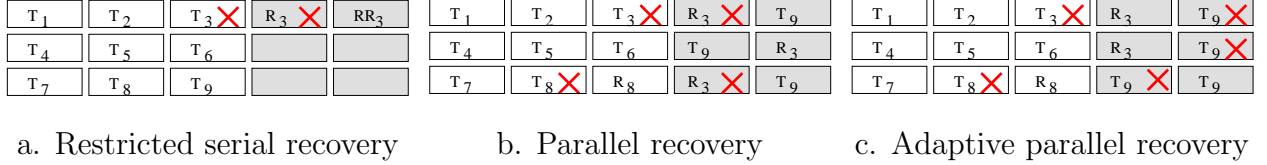


Figure 5.13: The maximum number of faults that can be tolerated by different recovery schemes in the worst case.

5.3.1.2 Parallel Recovery If faulty tasks can be re-processed on multiple duplex groups in parallel, we can allocate multiple recovery sections to re-execute one faulty task concurrently. The parallel recovery scheme considers all recovery sections at the beginning of backup slots and equally allocate them to the remaining tasks. For the above example, there are 6 recovery section in total and each of the remaining tasks R_3 and T_9 gets three recovery sections. The schedule is shown in Figure 5.12b.

Suppose that there are i faults during primary execution and i tasks remain to be executed/re-executed at the beginning of the backup slots. With $b \cdot p$ recovery sections in total, each remaining task will get at least $\lfloor \frac{b \cdot p}{i} \rfloor$ recovery sections. That is, at most $\lfloor \frac{b \cdot p}{i} \rfloor - 1$ additional faults can be tolerated. Therefore, when there are i faults during primary execution, the number of additional faults that can be tolerated during the backup execution by parallel recovery is:

$$PR(b, p, i) = \left\lfloor \frac{b \cdot p}{i} \right\rfloor - 1 \quad (5.70)$$

Let $PR_{b,p}$ represents the maximum number of faults that can be tolerated by p duplex groups with b backup slots in the worst case. Hence:

$$PR_{b,p} = \min_{1 \leq i \leq \min\{b \cdot p, n \cdot p\}} \{i + PR(b, p, i)\} \quad (5.71)$$

Notice that, $w (= n \cdot p)$ is the maximum number of faults that could occur during the n primary time units. That is, $i \leq n \cdot p$. Furthermore, we have $i \leq b \cdot p$ because it is not feasible for $b \cdot p$ recovery sections to recover more than $b \cdot p$ faulty tasks. Algebraic manipulations show that the value of $PR_{b,p}$ is obtained when:

$$i = \min \left\{ n \cdot p, \left\lfloor \sqrt{b \cdot p} \right\rfloor + u \right\}. \quad (5.72)$$

where u equals 0 or 1 depending on the floor operation in Equation 5.70. For the example in Figure 5.12, we have $PR_{2,3} = 4$ when $i = 2$ (illustrated in Figure 5.13b) or $i = 3$. That is, for the case shown in Figure 5.13b, two more faults can be tolerated in the worst case. One additional fault could cause the third recovery section for R_3 to be faulty and lead to a system failure. Notice that, although T_9 is processed successfully during the first backup slot, the other two recovery sections in the second backup slot that are allocated to T_9 can not be used by R_3 due to the fixed recovery schedule.

5.3.1.3 Adaptive Parallel Recovery Instead of considering all recovery sections together, we can use *one* backup slot *at a time* and adaptively allocate the recovery sections to improve the performance and tolerate more faults. For example, as shown in Figure 5.12c, we first use the three recovery sections in the first backup slot to execute/re-execute the remaining two tasks. Task recovery R_3 is processed on two duplex groups and task T_9 on one duplex group. If the Duplex that executes task T_9 happens to encounter a fault, the recovery task R_9 can be processed using all recovery sections in the second backup slot on all three duplex groups, thus allowing two additional faults as shown in Figure 5.13c. Therefore, 5 faults can be tolerated in the worst case. Compared to the simple parallel recovery scheme, one more fault could be tolerated.

In general, suppose that there are i tasks remaining to be executed/re-executed before using backup slots. Since there are p recovery sections within one backup slot, we can use the first backup slot to execute/re-execute up to p remaining tasks. If $i > p$, the remaining tasks and any new faulty tasks during the first backup slot will be executed/re-executed on the following $b - 1$ backup slots. If $i \leq p$, tasks are executed/re-executed redundantly using

a round-ribbon scheduler. In other words, $p - i \lfloor \frac{p}{i} \rfloor$ tasks are executed with a redundancy of $\lfloor \frac{p}{i} \rfloor + 1$ and the other tasks are executed with a redundancy of $\lfloor \frac{p}{i} \rfloor$.

Assuming that z tasks need to be executed/re-executed after the first backup slot, then the same recovery algorithm that is used in the first backup slot to execute i tasks is used in the second backup slot to process z tasks; and the process is repeated for all b backup slots.

With adaptive parallel recovery scheme, suppose that $APR_{b,p}$ is the worst case maximum number of faults that can be tolerated using b backup slots on p duplex groups. We have:

$$APR_{b,p} = \min_{1 \leq i \leq \min\{b \cdot p, n \cdot p\}} \{i + APR(b, p, i)\} \quad (5.73)$$

where i is the number of faults during the n primary time units and $APR(b, p, i)$ is the maximum number of additional faults that can be tolerated during b backup slots in the worst case distribution of the faults.

In Equation 5.73, $APR_{b,p}$ is calculated by considering different number of faults, i , occurred in the n primary time units and estimating the corresponding number of faults allowed in the worst case in backup slots, $APR(b, p, i)$, and then taking the minimum over all values of i . Notice that at most $w = n \cdot p$ faults can occur during the primary execution of w tasks and at most $b \cdot p$ faults can be recovered with b backup slots. That is $i \leq \min\{n \cdot p, b \cdot p\}$. $APR(b, p, i)$ can be found iteratively as shown below:

$$APR(1, p, i) = \left\lfloor \frac{p}{i} \right\rfloor - 1 \quad (5.74)$$

$$APR(b, p, i) = \min_{x(i) \leq J \leq y(i)} \{J + APR(b - 1, p, z(i, J))\} \quad (5.75)$$

When $b = 1$ (i.e., $i \leq p$), Equation 5.74 says that the maximum number of additional faults that can be tolerated in the worst case is $\lfloor \frac{p}{i} \rfloor - 1$. That is, one more fault could cause a recovery failure that leads to a system failure since at least one task is recovered with redundancy $\lfloor \frac{p}{i} \rfloor$.

For the case of $b > 1$, in Equation 5.75, J is the number of faults during the first backup slot and $z(i, J)$ is the number of tasks that still need to be executed during the remaining $b - 1$ backup slots. We search all possible values of J and the minimum value of $J + APR(b - 1, p, z(i, J))$ is the worst case maximum number of additional faults that can be tolerated during b backup slots.

The bounds on J , $x(i)$ and $y(i)$, depend on i , the number of tasks that need to be executed during b backup slots. When $i > p$, we have enough tasks to be executed and the first backup slot is used to execute p tasks (each on one processing unit). When J ($0 \leq J \leq p$) faults happen during the first backup slot and the total number of tasks that remain to be executed during the remaining $b - 1$ backup slots is $z(i, J) = i - p + J$. Since we should have $z(i, J) \leq (b - 1)p$, then J should not be larger than $b \cdot p - i$. That is, when $i > p$, we have $x(i) = 0$, $y(i) = \min\{p, b \cdot p - i\}$ and $z(i, J) = i - p + J$.

When $i \leq p$, all tasks are processed during the first backup slot with the least redundancy being $\lfloor \frac{p}{i} \rfloor$. To get the maximum number of faults that can be tolerated, at least one recovery failure is needed during the first backup slot such that the remaining $b - 1$ backup slots can be utilized. Thus, the lower bound for J , the number of faults during the first backup slot, is $x(i) = \lfloor \frac{p}{i} \rfloor$. Therefore, $\lfloor \frac{p}{i} \rfloor = x(i) \leq J \leq y(i) = p$. When there are J faults during the first backup slot, the maximum number of recovery failures in the worst case is $z(i, J)$, which is also the number of tasks that need to be executed during the remaining $b - 1$ backup slots. From the adaptive parallel recovery scheme, it is not hard to get $z(i, J) = \left\lfloor \frac{J}{\lfloor \frac{p}{i} \rfloor} \right\rfloor$ when $\lfloor \frac{p}{i} \rfloor \leq J \leq (i - p + i \lfloor \frac{p}{i} \rfloor) \lfloor \frac{p}{i} \rfloor$ and $z(i, J) = (i - p + i \lfloor \frac{p}{i} \rfloor) + \left\lfloor \frac{J - (i - p + i \lfloor \frac{p}{i} \rfloor) \lfloor \frac{p}{i} \rfloor}{\lfloor \frac{p}{i} \rfloor + 1} \right\rfloor$ when $(i - p + i \lfloor \frac{p}{i} \rfloor) \lfloor \frac{p}{i} \rfloor < J \leq p$.

For the example in Figure 5.12, applying Equations 5.74 and 5.75, we get $APR(2, 3, 1) = 5$. That is, if there is only one fault during the primary execution, it can tolerate up to 5 faults since all 6 recovery sections will be redundant. Similarly, $APR(2, 3, 2) = 3$ (illustrated in Figure 5.13c), $APR(2, 3, 3) = 2$, $APR(2, 3, 4) = 1$, $APR(2, 3, 5) = 0$ and $APR(2, 3, 6) = 0$. Thus, from Equation 5.73, $APR_{2,3} = \min_{i=1}^6 \{APR(2, 3, i) + i\} = 5$.

5.3.1.4 Arbitrary Number of Tasks We have discussed the case where the number of tasks, w , in an application is a multiple of p , the number of working duplex groups. Next, we focus on extending the results to the case where w is *not* a multiple of p .

Without loss of generality, suppose that $w = n \cdot p + d$, where $n \geq 1$ and $0 < d < p$. Thus, the execution of all tasks will need $(n + 1)$ primary time units. However, the last primary time unit is not fully scheduled with tasks. If we consider the last primary time unit as a backup slot, there will be *at least* d tasks that need to be executed after finishing

the execution in the first n time units.

Therefore, similar to Equations 5.70 and 5.73, the worst case maximum number of faults that can be tolerated with b backup slots can be obtained as:

$$PR_{b+1,p} = \min_{d \leq i \leq \min\{w, (b+1) \cdot p\}} \{i + PR(b+1, p, i)\} \quad (5.76)$$

$$APR_{b+1,p} = \min_{d \leq i \leq \min\{w, (b+1) \cdot p\}} \{i + APR(b+1, p, i)\} \quad (5.77)$$

where i is the number of tasks to be executed/re-executed on $b+1$ backup slots. $PR(b+1, p, i)$ and $APR(b+1, p, i)$ are defined as in Equations 5.70 and 5.75, respectively. That is, we pretend to have $b+1$ backup slots and treat the last d tasks that are not scheduled within the first n time units as faulty tasks. Therefore, the minimum number of faulty tasks to be executed/re-executed is d and the maximum number of faulty tasks is $\min\{w, (b+1) \cdot p\}$, which are shown as the range of i in Equations 5.76 and 5.77.

5.3.1.5 Maximum Number of Tolerated Faults To illustrate the performance of different recovery schemes, we calculate the worst case maximum number of faults that can be recovered by p duplex groups with b backup slots under different recovery schemes. Recall that, for the restricted serial recovery scheme, the number of faults that can be tolerated in the worst case is the number of available backup slots b and is independent of the number of duplex groups that work in parallel.

Table 5.1: The worst case maximum number of faults that can be tolerated by p duplex groups with b backup slots.

b		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	20
$p = 4$	parallel	3	4	6	7	8	8	9	10	11	11	12	12	13	14	14	16
	adaptive	3	6	10	14	18	22	26	30	34	38	42	46	50	54	58	78
$p = 8$	parallel	4	7	8	10	11	12	14	15	16	16	17	18	19	20	20	24
	adaptive	4	10	17	24	31	39	47	55	63	71	79	87	95	103	111	151
$p = 12$	parallel	6	8	11	12	14	16	17	18	19	20	22	23	24	24	25	30
	adaptive	6	14	23	33	43	54	66	78	90	102	114	126	138	150	162	222

Assuming that the number of tasks w in an application is a multiple of p and is more than the number of available recovery sections, Table 5.1 gives the worst case maximum number

of faults that can be tolerated by a given number of duplex groups with different numbers of backup slots under parallel and adaptive parallel recovery schemes. From the table, we can see that the number of faults that can be tolerated by the parallel recovery scheme may be less than what can be tolerated by the restricted serial recovery scheme. For example, with $p = 4$, the restricted serial recovery scheme can tolerate 15 and 20 faults when $b = 15$ and $b = 20$, respectively. However, the parallel recovery can only tolerate 14 and 16 faults respectively. The reason comes from the unwise decision of fixing the allocation of all the recovery slots, especially for larger number of backup slots. When the number of backup slots equals 1, the two parallel recovery schemes have the same behavior and can tolerate the same number of faults.

From Table 5.1, we can also see that the adaptive parallel recovery scheme is much more efficient than the restricted serial recovery and the simple parallel recovery schemes, especially for more duplex groups that work in parallel and larger number of backup slots. Interestingly, for the adaptive parallel recovery scheme, the number of faults that can be tolerated by p duplex groups increases linearly with the number of backup slots b when b is greater than a certain value that depends on p . For example, with $p = 8$, after b is greater than 5, the number of faults that can be tolerated using adaptive parallel recovery scheme increases by 8 when b is incremented. However, for $p = 12$, when $b > 7$, the number of faults increases by 12 when b is incremented.

5.3.2 Parallel Recovery and Modular Redundancy

Instead of being configured with Duplex, a system that consists of M processing units can be configured with any level of modular redundancy (e.g., NMR, $2 \leq N \leq M$), which may consume different amounts of energy and result in different numbers of faults that can be tolerated within an application's deadline. In this section, we extend the recovery schemes and combine them with modular redundancy.

Suppose that a section in Figure 5.11, 5.12 and 5.13 corresponds to the execution of a task on a NMR group. As discussed in Section 5.2, for the general a -out-of- N NMR model, at least a processing units in a NMR group need to get correct results to avoid a *NMR group*

failure. If there is a NMR group failure, we will get a faulty task. As in the last section, the faulty task can be re-executed using recovery sections to increase system performability.

For the discussion in Section 5.3.1, if each rectangle in Figure 5.12 and 5.13 corresponds to the execution of a task on a NMR group, the worst case maximum number of faults that can be tolerated by p duplex groups with b backup slots will be equal to the worst case maximum number of group failures that can be tolerated by p NMR groups with b backup slots.

Notice that, when N -modular redundancy ($2 \leq N \leq M$) is used, the number of faulty sections to be recovered may not be the same as the number of faults to be tolerated. For the general a -out-of- N NMR model, the execution of one task succeeds if no less than a processing units get correct results. That is, in the worst case, $N - a + 1$ faults cause a NMR group failure and thus lead to a faulty task. Therefore, to tolerate k faults, we only need to be able to recover $\lfloor \frac{k}{N-a+1} \rfloor$ faulty tasks. Inversely, if Q is the number of faulty sections that can be recovered, the number of faults that can be tolerated will be $Q \cdot (N - a + 1) + N - a$. For example, if each rectangle in Figure 5.13c represents the execution of a task on a TMR and the 2-out-of-3 TMR model is used, the maximum number of faults that can be tolerated in the worst case will be $APR(2, 3) \cdot (N - a + 1) + N - a = 5(3 - 2 + 1) + 3 - 2 = 11$.

5.3.3 Optimal Redundant Configurations

In what follows, we consider two optimization problems. First, for a given performability goal, what is the optimal redundant configuration that minimizes system energy consumption? Second, for a limited energy budget, what is the optimal redundant configuration that maximizes system performability? A *redundant configuration* is defined as the number processing units deployed, the level of modular redundancy used, the processing frequency for working processing units and the number of backup time slots needed.

5.3.3.1 Minimize Energy with A Given Performability Goal For a system consisting of M processing units, to tolerate k faults within an application's deadline, we may use different redundant configurations which in turn consume different amounts of energy.

When the system is configured with NMR, there are at most $\lfloor \frac{M}{N} \rfloor$ NMR groups available. Because of energy consideration, as discussed in Section 4.7, it may be more energy efficient to use fewer NMR groups than what is available and turn the unused processing units off.

For p NMR groups ($1 \leq p \leq \lfloor \frac{M}{N} \rfloor$; the remaining $M - pN$ processing units are turned off for energy savings), we have shown in Sections 5.3.1 and 5.3.2 how to compute the maximum number of faults, k , that can be tolerated by p NMR groups with a given number of backup time slots b in the worst case. In this section, we use the same analysis as discussed for the inverse problem. That is, to compute the *least* number of backup slots, b , needed by p NMR groups to tolerate k faults.

If the backup time needed is more than the slack time available (i.e., $b > D - \lceil \frac{w}{p} \rceil$), p NMR groups are not *feasible* to tolerate k faults within the application's deadline. Suppose that $b \leq D - \lceil \frac{w}{p} \rceil$, the amount of remaining slack time on each NMR group is $D - \lceil \frac{w}{p} \rceil - b$. As in Section 5.1.1, expecting k_e faults will occur (i.e., k_e -pessimism) and assuming that b_e ($\leq b$) is the minimum number of backup slots to tolerate k_e faults, the slack time is used to scale down the processing frequency of the execution during $\lceil \frac{w}{p} \rceil$ primary time units as well as the first b_e backup slots. The recoveries during the remaining backup slots are executed at the maximum frequency f_{max} when more than k_e faults occur. Thus, the *k_e -pessimism expected energy consumption* is:

$$f(k_e) = \left\{ \frac{\lceil w/p \rceil + b_e}{D - (b - b_e)}, f_{ee} \right\} \quad (5.78)$$

$$E(k_e) = p \cdot N \cdot \left[P_s D + (P_{ind} + C_{ef} f^m(k_e)) \frac{\lceil \frac{w}{p} \rceil + b_e}{f(k_e)} \right] \quad (5.79)$$

where $f(k_e)$ is the frequency to execute tasks during the primary time units and the first b_e backup slots. Recall that f_{ee} is the minimum energy efficient frequency (see Section 3.2.2).

At last, searching through all feasible redundant configurations, Algorithm 3 summarizes the procedure to get the optimal redundant configuration to minimize the expected energy consumption while tolerating k faults within the application's deadline D .

First, the algorithm finds the least number of backup slots b for p NMR groups to tolerate k faults using a given recovery scheme (lines 7 and 8). If b is larger than the available slack $D - n$, p NMR groups are not feasible. Otherwise (from line 11 to 17), the least number

Algorithm 3 The optimal redundancy configuration algorithm for energy minimization

```
1: INPUT:  $w, D, M, \alpha, \beta, \kappa, m, k, k_e$ 
2:  $E_{min} = M(\alpha + \beta + 1)P_d^{max}D$ ;
3:  $N^{opt} = -1$ ;  $p^{opt} = -1$ ;  $b^{opt} = -1$ ;
4: for ( $N$  from 2 to  $M$ ) do
5:   for ( $p$  from  $\lfloor \frac{M}{N} \rfloor$  to 1) do
6:      $b = 0$ ;  $b_e = 0$ ;
7:     while ( $PR(N, c, p, b) < k$ ) do
8:        $b = b + 1$ ; /*see Equations 5.76 and 5.77*/
9:     end while
10:    if ( $b \leq D - \lceil \frac{w}{p} \rceil$ ) then
11:      while ( $PR(N, c, p, b_e) < k_e$ ) do
12:         $b_e = b_e + 1$ ;
13:      end while
14:      Calculate  $E(k_e)$  from Equation 5.79;
15:      if ( $E(k_e) < E_{min}$ ) then
16:         $E_{min} = E(k_e)$ ;  $N^{opt} = N$ ;  $p^{opt} = p$ ;  $b^{opt} = b$ ;
17:      end if
18:    end if
19:  end for
20: end for
21: return  $(N^{opt}, p^{opt}, b^{opt})$ .
```

of backup slots b_e for p NMR groups to tolerate k_e faults is obtained (line 11) and the expected energy consumption is computed (line 14). Searching through all feasible numbers of NMR groups (line 5) and all possible values of N (line 4), we get the optimal redundant configuration to minimize the expected energy consumption (line 21). Notice that, finding the least number of backup slots to tolerate k faults has a complexity of $\mathbf{O}(k)$ (lines 7 and 11). Thus, the complexity of this algorithm is $\mathbf{O}(M^2k)$.

5.3.3.2 Maximize Performability with Fixed Energy Budget When the energy budget is limited, we may not be able to power up all the M processing units in a system. The larger the number of deployed processing units is, the lower the frequency at which the processing units can run. For a given number of processing units that run at a certain frequency, different levels of modular redundancy will result in different number of modular redundancy groups and further lead to different maximum number of faults that can be tolerated within an application's deadline. In this section, we consider the optimal redundant configuration that maximizes the number of faults that can be tolerated with fixed energy

budget.

Recall that, it is most energy efficient to scale down all the deployed processing units uniformly within an application's deadline (from the power model discussed in Section 3.2). With limited energy budget E_{budget} and an application's deadline D , the maximum power level that a system can consume is:

$$P_{budget} = \frac{E_{budget}}{D} \quad (5.80)$$

When p NMR groups ($p \cdot N \leq M$) is deployed, the minimum power level is consumed when every processing unit runs at the minimum energy efficient frequency f_{ee} . Thus, the minimum power level is:

$$P_{min}(p, N) = p \cdot N(P_s + P_{ind} + C_{ef}f_{ee}^m) = p \cdot N(\alpha + \beta + \kappa^m)P_d^{max} \quad (5.81)$$

If $P_{min}(p, N) > P_{budget}$, p NMR groups are not feasible in terms of energy consumption. Suppose that $P_{min}(p, N) < P_{budget}$, which means that the processing units in p NMR groups could run at a higher frequency than f_{ee} . Assume that the frequency is $f_{budget}(p, N)$. We have:

$$f_{budget}(p, N) = \sqrt[m]{\frac{P_{budget}}{p \cdot N \cdot P_d^{max}} - \alpha - \beta} \quad (5.82)$$

The total time needed for executing all tasks in an application at frequency $f_{budget}(p, N)$ is:

$$t_{primary} = \frac{\left\lceil \frac{w}{p} \right\rceil}{f_{budget}(p, N)} \quad (5.83)$$

If $t_{primary} > D$, p NMR groups are not feasible to finish all tasks within the application's deadline under the energy budget. Suppose that $t_{primary} \leq D$. We have $D - t_{primary}$ units of slack time and the number of backup time slots that can be scheduled at frequency $f_{budget}(p, N)$ is:

$$b_{budget}(p, N) = (D - t_{primary})f_{budget}(p, N) = D \cdot f_{budget}(p, N) - \left\lceil \frac{w}{p} \right\rceil \quad (5.84)$$

Thus, from Equations 5.76 and 5.77 in Section 5.3.1, the maximum number of NMR group failures that can be recovered within the application's deadline is either $PR(p, b_{budget}(p, N))$

(for parallel recovery scheme) or $APR(p, b_{budget}(p, N))$ (for adaptive parallel recovery scheme).

The corresponding maximum number of faults that can be tolerated in the worst case is:

$$k_{parallel}(p, N) = PR(p, b_{budget}(p, N))(N - a + 1) + (N - a) \quad (5.85)$$

$$k_{adaptive}(p, N) = APR(p, b_{budget}(p, N))(N - a + 1) + (N - a) \quad (5.86)$$

where a $a - out - N$ NMR model is assumed.

Algorithm 4 Algorithm to find the optimal redundant configuration for maximizing per-
formability

```

1: INPUT:  $n, D, M, \alpha, \beta\kappa, m, E_{limit}$ 
2:  $k_{max} = -1; N^{opt} = -1; p^{opt} = -1;$ 
3: Get  $P_{limit}$  from Equation 5.80;
4: for ( $N$  from 2 to  $M$ ) do
5:   for ( $p$  from  $\lfloor \frac{M}{N} \rfloor$  to 1) do
6:     Get  $f_{limit}(p, N)$  from Equation 5.82;
7:     Get  $t_{primary}$  from Equation 5.83;
8:     if ( $f_{limit}(p, N) \geq f_{ee}$  AND  $t_{primary} \leq D$ ) then
9:       Get  $b_{limit}(p, N)$  from Equation 5.84;
10:      Get  $k(p, N)$  from either Equation 5.76 or Equation 5.77;
11:      if ( $k(p, N) > k_{max}$ ) then
12:         $k_{max} = k(p, N); N^{opt} = N; p^{opt} = p;$ 
13:      end if
14:    end if
15:  end for
16: end for
17: return  $(N^{opt}, p^{opt})$ .
```

For a given recovery scheme, searching all feasible combinations of p and N , Algorithm 4 summaries the procedure to get the optimal redundant configuration that maximizes the maximum number of faults that can be tolerated within the application's deadline in the worst case. It is easy to find that the complexity of this algorithm is $\mathbf{O}(M^2)$.

5.3.4 Analysis Results

In what follows, we provide some analysis results to show that different levels of modular redundancy may be deployed in optimal redundant configurations that minimizes expected energy consumption or maximizes system performability. Assuming that any two faults are

not the same, the *2-out-of-N* NMR model is used, which actually favors higher levels of modular redundancy.

In our analysis, we focus on varying the task size, number of tasks in an application (i.e., system load), the number of faults to be tolerated (k) and the recovery schemes to see how they affect the optimal redundant configuration. The deadline of the application is assumed to be 1 second and three different task sizes are considered: $1ms$, $10ms$ and $50ms$.

As before, we use $m = 3$ and $P_d^{max} = C_{ef} f_{max}^m = 1$ in our analysis. The values of α and β are assumed to be 0.1 and 0.3 respectively. We consider a system that consists of 12 processing units. To detect faults through comparison, the least level of modular redundancy is Duplex and there are at most 6 duplex groups. Define *system load* as the ratio of the WCET of an application, which is the summation of all tasks' WCET, over its deadline. With 6 duplex groups, the maximum system load that can be handled is 6. To get enough slack for illustrating the variation of the optimal redundant configurations, a system load of 2.6 is used. For different task sizes, different numbers of tasks in the application are used to obtain system load as 2.6.

5.3.4.1 Optimal Configuration for Energy Minimization Table 5.2 shows the level of modular redundancy employed (N) and the number of NMR groups used (p). for the optimal redundant configuration that tolerates a given number of faults k using different recovery schemes (the remaining processing units are turned off for energy efficiency).

Table 5.2: The optimal redundant configuration for different task sizes with fixed system load 2.6 using different recovery schemes. Here, $M = 12$ and $k_e = \frac{k}{2}$.

k		1	2	3	4	5	6	7	8	9	10	11	12	13	14
	size,number	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p
serial	1ms,2600	2, 4	2, 4	2, 4	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5
	10ms,260	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5
	50ms,52	2, 4	2, 4	2, 4	2, 6	2, 6	2, 6	2, 6	2, 6	2, 6	2, 6	3, 4	3, 4	3, 4	3, 4
parallel	1ms,2600	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5
	10ms,260	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5
	50ms,52	2, 4	2, 5	2, 5	2, 5	2, 5	2, 6	2, 6	2, 5	2, 5	2, 5	2, 5	2, 5	2, 6	3, 4
adaptive	1ms,2600	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 5	2, 5	2, 5	2, 5
	10ms,260	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5
	50ms,52	2, 4	2, 5	2, 5	2, 5	2, 5	2, 4	2, 4	2, 5	2, 5	2, 5	2, 6	2, 5	2, 5	2, 6

From the table, we can see that Duplex is the most energy efficient configuration in most

cases. Moreover, small tasks and adaptive parallel recovery favor lower levels of modular redundancy while large tasks and restricted serial recovery favor higher levels of modular redundancy, especially for larger number of faults to be tolerated (more than 11 in the table). Due to the effects of sleep power, the number of duplex groups needed does not increase monotonically when the number of faults increases, especially for the case of large task size where more slack time is needed as temporal redundancy for the same number of backup slots. Figure 5.14 shows the corresponding minimum expected energy consumption. Here, $\frac{k}{2}$ -*pessimism* is used.

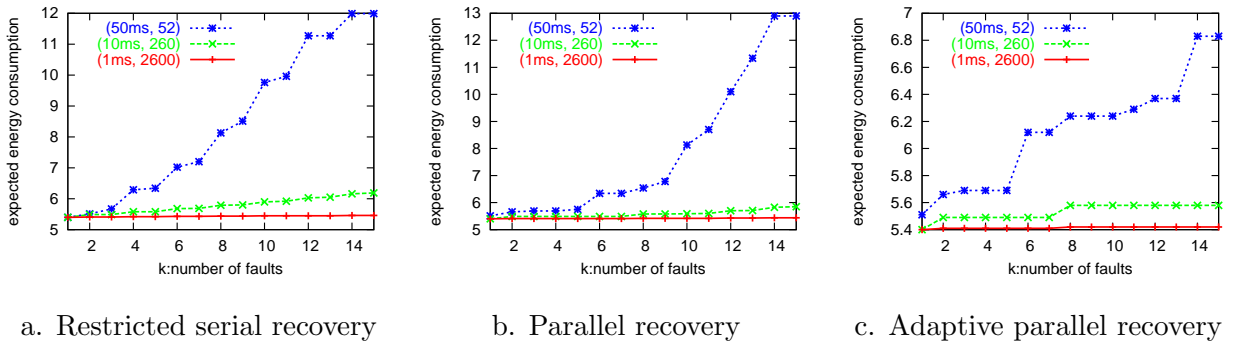


Figure 5.14: The minimum expected energy consumption for different recovery schemes.

The two numbers in the legends stand for task size and the number tasks in the application, respectively. From the figure, we can see that, when the task size is $1ms$, the minimum expected energy consumption is almost the same for different numbers of faults to be tolerated. The reason is that, to tolerate up to 15 faults, the amount of slack time used by the backup slots is almost negligible and the amount of slack time used for energy management is more or less the same when each backup slot is only $1ms$. However, when the task size is $50ms$, the size of one backup slot is also $50ms$ and the minimum expected energy consumption increases significantly when the number of faults to be tolerated increases. This comes from the fact that each additional backup slot needs relatively more slack time and less slack is left for energy management when the number of faults to be tolerated increases.

Notice that, for different number of faults to be tolerated, the same number of backup slots may be needed, especially for parallel recovery schemes, which in turn leads to the same minimum expected energy consumption. Furthermore, to tolerate the same number of

faults, the adaptive parallel recovery scheme is the most energy efficient for a give task size.

For different task sizes and the adaptive parallel recovery scheme, Figure 5.15 further shows the minimum expected energy consumption to tolerate given numbers of faults under different system loads. For a given task size, different numbers of tasks are used to obtain different system loads.

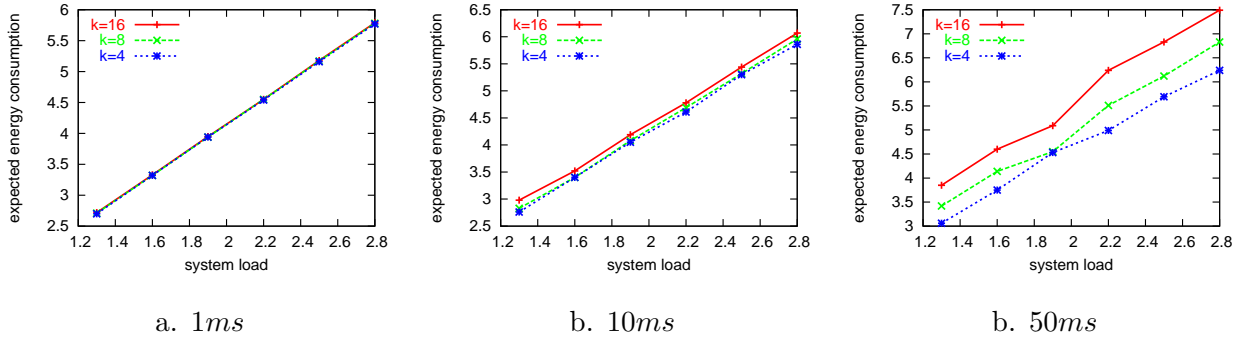


Figure 5.15: The minimum expected energy consumption under different system load for different task sizes to tolerate given numbers of faults. The adaptive parallel recovery scheme is used and $k_e = \frac{k}{2}$.

When system load increases, more tasks need to be processed within the application's deadline and the minimum expected energy consumption to tolerate a given number (e.g., 4, 8 or 16) of faults increases. The same as before, when the task size is 1ms, the minimum expected energy consumption is almost the same to tolerate 4, 8 or 16 faults within the application's deadline. The difference in the minimum expected energy consumption increases for larger task sizes.

To see the effects of different levels of pessimism, Table 5.3 shows the optimal redundant configuration that minimizes the expected energy consumption when tolerating given numbers of faults. For adaptive parallel recovery, Duplex is always the best and is not shown in the table. From the table, we can see that optimistic analysis favors lower levels of modular redundancy and pessimistic analysis favors higher levels of modular redundancy. This comes from the fact that higher levels of modular redundancy needs less backup slots for tolerating a given number of faults, which results in more slack for scaling down all the processing and less energy when pessimistic analysis is used. For optimistic analysis, only the original

Table 5.3: The effects of pessimism levels on optimal redundant configuration to tolerate a given number of faults. The task size used is $50ms$ and there are 26 tasks in the application.

k		1	2	3	4	5	6	7	8	9	10	11	12	13	14
recovery	k_e	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p	N, p
restricted	0	2, 2	2, 2	2, 3	2, 3	2, 3	2, 3	2, 3	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5
	$k/2$	2, 2	2, 2	2, 3	2, 3	2, 3	2, 3	2, 3	2, 3	2, 5	3, 3	3, 3	3, 3	3, 3	3, 3
	k	2, 2	2, 2	2, 2	2, 3	2, 3	2, 3	3, 2	3, 3	3, 3	3, 3	3, 3	3, 3	3, 3	4, 3
parallel	0	2, 2	2, 2	2, 2	2, 3	2, 3	2, 3	2, 3	2, 3	2, 5	2, 5	2, 5	2, 5	2, 5	2, 5
	$k/2$	2, 2	2, 2	2, 2	2, 3	2, 3	2, 3	2, 4	2, 3	2, 4	2, 5	2, 5	2, 5	2, 5	3, 4
	k	2, 2	2, 2	2, 2	2, 3	2, 3	2, 3	2, 4	2, 5	3, 3	3, 3	3, 3	3, 3	3, 3	4, 3

processing of tasks counts in the expected energy consumption, which favors lower levels of modular redundancy that may have more backup slots.

5.3.4.2 Optimal Configuration for Performability Maximization Assume that the maximum power P_{max} corresponds to running all processing units with the maximum processing frequency f_{max} . When the energy budget for each interval is limited, we can only consume a fraction of P_{max} when processing tasks within an application's deadline. For different energy budget (i.e., different fraction of P_{max}), Figure 5.16 shows the worst case maximum number of faults that can be tolerated when the optimal redundant configuration is employed (see Algorithm 4). Again, different numbers of tasks are considered for different task sizes to get a fixed system load of 2.6.

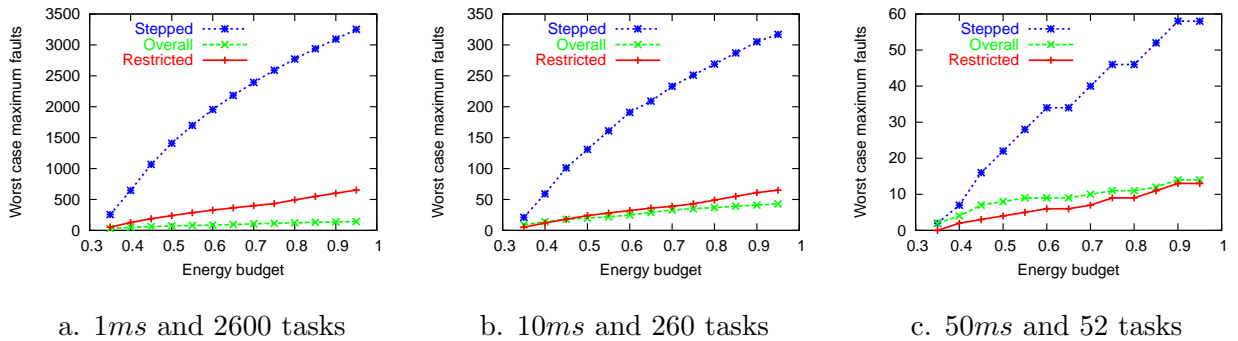


Figure 5.16: The worst case maximum number of faults that can be tolerated with limited energy budget for different sizes of tasks.

From the figure, we can see that the number of faults that can be tolerated increases

with increased energy budget. When the task size increases, there is less available backup slots due to large backup slot size and fewer faults can be tolerated. When the number of backup slots is very large (e.g., for the case of task size being $1ms$ with 2600 tasks), the parallel recovery performs worse than the restricted serial recovery. The adaptive parallel recovery performs the best and can tolerate much more faults than the other two recovery schemes at the expense of more complex management of backup slots. For optimal redundant configurations, similar results have been observed regarding the relationship of the levels of modular redundancy, recovery schemes and task sizes.

5.4 INTERPLAY OF ENERGY MANAGEMENT AND PERFORMABILITY

So far we have assumed that the fault rate is independent of energy management. However, the rate of transient faults (i.e., soft errors caused, for example, by cosmic ray radiations) do depend on system processing frequency and supply voltage as discussed below. This makes the trade-off between system performability and energy consumption more interesting. For simplicity, when considering the effects of voltage scaling on fault rates, we re-visit the single task applications and checkpointing techniques for a duplex system as in Section 5.1. However, similar analysis may be performed for applications consisting of multiple tasks and systems with higher levels of modular redundancy.

Based on previously published data, we first discuss an exponential fault rate model for different supply voltages (Section 5.4.1). Then, the trade-off between energy consumption and system performability for single task applications running on a duplex system is explored and the analysis results are presented (Section 5.4.2).

5.4.1 Voltage Scaling and Fault Rates

Transient faults caused by radiations in semiconductor circuits have been known and well studied since the late 1970s [101]. When high-energy particles strike a sensitive region in semiconductor devices, a dense track of electron-hole pairs is deposited. The charge may

be collected by pn-junctions via drift and diffusion mechanisms to form a current pulse and cause a logic error [41], or to accumulate and exceed the minimum charge (i.e., the *critical charge*) required to flip the value stored in a memory cell [32, 102].

Because it is relatively easy to detect errors in memory and large areas in microprocessor chips is dedicated to caches and registers, numerous work has examined the effects of cosmic ray radiations on memory circuits [32, 78, 102]. Although past research has shown that logic circuits are less susceptible to cosmic ray radiations than memory [12, 52], a recent model predicts that, with technology advancement and reduced feature size, the fault rate in combinational logic circuits will be comparable to that of memory elements [81]. There are various factors that affect the fault rate, such as cosmic ray flux (i.e., number of particles per area), technology feature size, chip capacity, supply voltage and operating frequency. Thus, modeling the fault rate is extremely hard [78, 81, 102].

In this work, we focus on the effects of frequency and voltage scaling on fault rate changes. We assume that the radiation induced faults follow a Poisson distribution with an average fault rate λ being determined by system supply voltage and frequency. For simplicity, no variation of other factors is considered. That is, for a given level of supply voltage and frequency (e.g., V_{max} and f_{max}), the average fault rate (e.g., λ_0) is fixed. Hence, for systems running at frequency f ($f_{min} \leq f \leq f_{max}$) and corresponding voltage V ($V_{min} \leq V \leq V_{max}$), the general model for the average fault rate can be expressed as:

$$\lambda(f, V) = \lambda_0 \cdot g(f, V) \quad (5.87)$$

where λ_0 is the average fault rate corresponding to V_{max} and f_{max} . That is, $g(f_{max}, V_{max}) = 1$.

In what follows, we consider an exponential model for the fault rate based on previously published data. However, the framework for exploring the trade-off between system performability and energy consumption discussed in Section 5.4.2 is independent of specific fault rate models.

5.4.1.1 Exponential Fault Rate Model For different technologies that have different supply voltages, Seifert *et al.* examined the fault rate in the family of Alpha processors due to α particle effects using both simulations and experiments [77]. Their results showed that fault rate in these processors (including logic core and cache) increases exponentially when supply voltage decreases. The same observation has been shown in [102] for memory. The reason is that, with reduced supply voltage, the critical charge becomes smaller and in turn results in exponentially increased fault rate [32, 81]. This partially comes from the fact that there are many more lower energy particles than higher energy particles (e.g., one order of magnitude less in energy corresponding to 100 times more in the number of particles [101]) and with smaller critical charge, lower energy particles could cause an error.

As discussed in Section 3.2, voltage scaling reduces supply voltage for lower frequencies [66]. We use the conclusions in [32, 77, 81, 101, 102] to formulate the effects of voltage scaling on fault rates as an exponential model. That is, at the lowest frequency f_{min} and supply voltage V_{min} , the average fault rate is assumed to be $\lambda_{max} = \lambda_0 10^d$, where $d (> 0)$ is a constant. When a system runs at frequency f and corresponding voltage $V = f \cdot V_{max} = f$, the average fault rate can be expressed as (recall that normalized frequency and voltage are used):

$$\lambda(f, V) = \lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} \quad (5.88)$$

Therefore, in this model, reducing the supply voltage for lower frequency results in exponentially increased fault rates and larger d indicates that the fault rate is more sensitive to voltage scaling. Notice that, this is a pessimistic model. Considering that the fault rate generally decreases for lower frequencies and taking the thermal effects of energy management into consideration, it may be the case that fault rates do not increase exponentially with reduced voltages. We will explore these points in our future work.

5.4.2 Trade-off between Energy and Performability

For an application with system load as $\sigma = \frac{L}{D}$, where L is the WCET and D is the deadline, from the discussion in Section 5.1, the frequency at which an application should execute

can be as low as σf_{max} (if $\sigma > \kappa$) for the maximum energy savings. However, when the application is executed at frequency σf_{max} , there is no slack for recovery which leads to the minimum system performability. If the application is executed at the maximum frequency f_{max} , from Equation 5.12, the maximum number of recovery sections is b_{max} which leads to the highest system performability but no energy saving.

There is a trade-off between energy consumption and system performability for executing an application. The higher the processing frequency to execute an application is, the more energy is consumed. However, more recovery sections can be scheduled and higher level of performability is expected.

In this section, considering the effects of voltage scaling on fault rates, we explore the trade-off between energy consumption and system performability by assuming that an application is executed at different frequencies between σf_{max} and f_{max} . Suppose that an application is executed at frequency f ($\sigma f_{max} < f < f_{max}$) and corresponding supply voltage v on a duplex system. For simplicity, we assume that the application's recovery sections are executed at the same frequency as f (i.e., being pessimistic on the expected energy consumption).

From Equation 5.27, the maximum number of recovery sections $b_{max}(f)$ and the corresponding optimal number of checkpoints $n_{opt}(f)$ can be obtained. Therefore, the length of each section is:

$$t_{section} = \frac{L/n_{opt}(f)}{f} \quad (5.89)$$

Notice that, due to the integer limitation of $b_{max}(f)$ and $n_{opt}(f)$, different values of f may lead to the same number of recovery sections and the same number of optimal checkpoints within an application.

Assume that the average fault arrival rate is $\lambda(f, v)$ (see Equation 5.88), the probability of having fault(s) during the execution of one section on a duplex system is:

$$\rho_{section} = 1 - e^{-\lambda(f, v)t_{section}} \quad (5.90)$$

From Equation 5.35, 5.36 and 5.38, the expected energy consumption for executing the application at frequency f can be obtained.

5.4.2.1 Some Numeric Results To illustrate the significance of fault rate changes on performability under voltage scaling, we consider an application that has a deadline $D = 100$ and WCET $L = 30$. The checkpoint overhead is set as $r = 4$. With $P_d^{max} = 1$, $\alpha = 0.1$ and $\beta = 0.2$, we have $f_{ee} \approx 0.46$. We vary the values of d (0, 2, 4 and 6) and the results are shown in Figure 5.17. For other values of L (e.g., 40 and 50), similar results have been obtained.

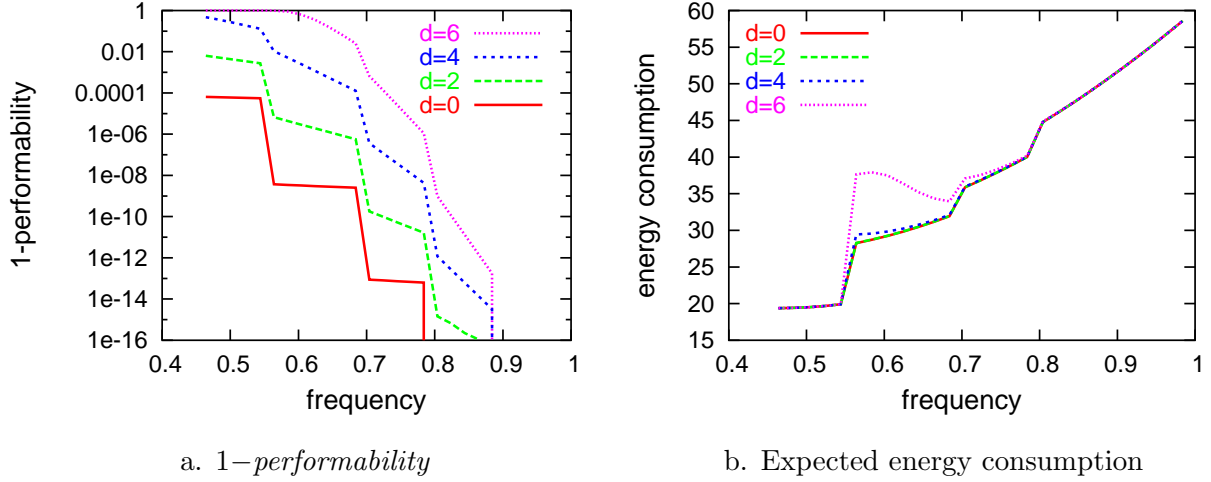


Figure 5.17: The performability and expected energy consumption for different values of d .

We assume that $\lambda_0 = 10^{-6}$. Recall that the fault rate under frequency f and corresponding voltage v is given in Equation 5.88. For a given level of frequency and voltage, the performability is obtained by employing the optimal number of checkpoints that maximizes the number of recovery sections. From Figure 5.17a, we can see that the larger the value of d is (i.e., the faster the fault rate increases when frequency and voltage decrease), the worse the performability becomes for lower frequencies. The sharp step increases in the probability of failure (i.e., decreases in the performability) correspond to one fewer recovery section being scheduled because of reduced frequencies. In general, the number of recovery sections determines the level of performability. The performability for fixed fault rate ($d = 0$) is much better than that of variable fault rates (e.g., better than the case of $d = 2$ in at least 2 orders of magnitude).

Figure 5.17b shows the corresponding expected energy consumption for different values of d . The expected energy consumption generally decreases with lower frequency/voltage,

which illustrates the trade-off between performability and expected energy consumption. However, for the case of $d = 6$, the expected energy consumption increases with reduced frequency when $0.58 \leq f \leq 0.68$. The anomaly comes from the high probability of recovery sections being executed with $d = 6$, which overshadows the energy saved during original execution. The sharp decrease around $f = 0.54$ is due to the fact that no recovery can be scheduled and that no checkpoint overhead is incurred.

The results show the implications of the applicability of voltage scaling when performability is a major concern. For example, if the required performability goal is $1 - 10^{-8}$, the frequency (and the corresponding voltage) can be reduced to 0.56 when ignoring the effects of voltage scaling on fault rates. However, when considering fault rate changes caused by voltage scaling, we can only reduce the frequency (and the corresponding voltage) to 0.7 when $d = 2$ or 0.78 when $d = 4$. Thus, ignoring the effects of voltage scaling on fault rates [89, 98] is too optimistic and may lead to unsatisfied performability when voltage scaling is used for energy savings.

5.5 CHAPTER SUMMARY

In this section, we explore schemes for energy efficient fault tolerance and illustrate the interplay of energy consumption and performability in parallel real-time systems.

Specifically, to efficiently use the slack time in a system, we consider checkpointing techniques and explore the energy efficient roll-back recovery schemes. We propose the concept of pessimism level for the number of expected faults and explore the optimal number of checkpoints to minimize the expected energy consumption when multiple recovery sections are considered. We further explore the optimal number of checkpoints to maximize system performability with limited energy budget.

Next, extending the idea of OTMR [22], we consider an energy efficient optimistic modular redundancy scheme (ONMR). Expecting that faults are rare, ONMR turns off or scales down some processing units in a modular redundant system provided that they can catch up and finish the computation before an application's deadline if other processing units do encounter faults. We analyze the optimal frequency setting for all processing units to minimize

the energy consumption in ONMR. The energy consumption and system performability are compared between ONMR and traditional modular redundancy schemes.

For an application being executed on a system that consists of a certain number of processing units, we address the optimal redundant configuration problems. The efficient parallel recovery schemes are first proposed. Then, a framework is discussed for finding the optimal redundant configuration to minimize expected energy consumption with a given performability goal as well as to maximize system performability with limited energy budget.

Finally, we address the trade-off between energy consumption and system reliability while considering the effects of energy management on fault rates. Based on previous published data, an exponential fault rate model for voltage scaling is proposed. A general framework to explore the trade-off between energy consumption and system performability is considered. We illustrate that ignoring the effects of voltage scaling on fault rates may be too optimistic and will lead to unsatisfied system performability when voltage scaling is used for energy savings.

6.0 CONCLUSIONS

Timeliness is as crucial as correctness of the results produced in real-time systems. For mission critical applications, such as space-based control systems or life maintenance systems, where a failure may cause catastrophic results, reliability is as important as timeliness. As real-time systems are generally over-designed and the actual execution time of real-time applications has great variations, slack time is expected and can be used by energy management schemes for energy savings as well as by roll-back recovery schemes for increasing system reliability. This work focuses on frame-based real-time applications and explores schemes for energy efficient fault tolerance in parallel systems.

We considered the traditional AND-model applications, where all tasks will be executed during any execution of an application, as well as the AND/OR-model applications, where only a subset of tasks will be executed during any specific execution of an application. As part of this dissertation, energy management schemes that explore both static and dynamic slack time in a system are first proposed. The proposed schemes demonstrate the feasibility of energy management in parallel systems and are shown to save a significant amount of energy through simulations. Then, when considering simultaneous management of energy and reliability, the interplay between energy consumption and reliability is addressed and a few energy efficient fault tolerance schemes are proposed to demonstrate the trade-off between energy savings and reliability and to illustrate the importance of managing them together.

Specifically, a *static power management with parallelism (SPM-P)* scheme is proposed. This scheme takes a schedule's parallelism into consideration while carrying out energy management. Compared to the scheme of *uniform static power management (U-SPM)*, which is optimal for uniprocessor systems and distributes static slack over a schedule evenly, SPM-P

can save 5% more energy when the energy consumed by *no power management (NPM)* is used as a baseline.

In addition to the SPM-P static power management scheme, *shifted/shared slack reclamation (S/SSR)* schemes that share slack among processing units in a system are proposed for managing the slack that is generated at run time. *Speculation schemes* are also addressed for the same purpose with the intent to smooth the frequency changes and save more energy by exploring the statistical timing information about applications. Simulation results show that, compared to static power management, S/SSR can get up to 80% energy savings when the average over worst case execution time of an application is around 50%. With the capability of automatically balancing actual workload among processing units by sharing the dynamic slack, S/SSR performs better than the partition scheduling where each processing unit reclaims slack for energy savings individually.

In order to account for the overhead of frequency/voltage changes, a *slack reservation* scheme is proposed to efficiently incorporate the overhead into our energy management algorithms. From the simulations, when the overhead of frequency/voltage changes is relatively small compared to the size of tasks, the effects of such overhead on energy management schemes for energy savings are not significant. Moreover, the effects of discrete frequency/voltage levels and the minimum frequency are also addressed. When there are only a few (e.g., 4 to 6) frequency/voltage levels in a system and/or there is a minimal frequency limitation due to static/leakage power, S/SSR consumes energy comparable to the speculation schemes. When shared memory contention is considered, our energy aware scheduling algorithms become more conservative and may reject applications that are schedulable. However, for applications that have a feasible schedule under our energy aware scheduling algorithms, more energy savings are obtained.

Furthermore, we addressed the question of “*what is the best we can do*” for energy savings and developed some theoretic bounds for energy management schemes. Clearly, the amount of energy that could be saved depends on the total amount of slack, both static and dynamic, in a system. For systems with 100% load, no slack exists and no energy saving can be obtained by energy management schemes. When a system is not fully loaded, the lower the system load is, the more slack there is in the system and the higher the energy savings

can be.

When combining energy and reliability management, *checkpointing* techniques are explored to efficiently use slack time as temporal redundancy and increase system reliability. We propose the concept of *pessimism level* for the number of expected faults and explore the optimal number of checkpoints to minimize expected energy consumption when multiple recovery sections are considered. We further explore the optimal number of checkpoints to maximize system reliability with limited energy budget. Without deploying checkpoints, systems that explore roll-back recovery for increasing system reliability can only handle the cases with system load less than 50%. When checkpoints are deployed, more checkpoints imply smaller recovery sections but incur more overhead. Our analysis shows that the optimal number of checkpoints that minimizes system energy consumption increases when the overhead of checkpoints decreases. Moreover, with limited energy budget, the optimal number of checkpoints that results in the maximum number of recovery sections generally leads to the highest (or close to the highest) system reliability.

Extending the idea of an optimistic triple modular redundancy scheme (OTMR) [22], an *optimistic N-modular redundancy (ONMR)* scheme is addressed, which reduces energy consumption for traditional N-modular redundant systems without degrading system reliability. Expecting that faults are rare, ONMR turns off or scales down some processing units in a modular redundant system provided that they can catch up and finish the computation before an application's deadline if other processing units do encounter faults. We analyze the optimal frequency setting for all processing units to minimize the energy consumption and compare ONMR with the traditional NMR scheme on both energy consumption and system reliability. The analysis results show that ONMR do save energy compared to NMR, but at the expense of more complex frequency management. Moreover, by forcing some processing units to run faster, ONMR uses less time to execute an application and reduces the probability of having fault(s), thus achieves better system reliability than NMR.

For a fully application being executed on a system that consists of a given number of processing units, we address the *optimal redundant configuration* problems. Considering the parallelism of temporal redundancy in a system, an efficient *adaptive parallel recovery* scheme is first proposed. A framework is presented for finding the optimal redundant configuration

to minimize expected energy consumption with a given reliability goal as well as to maximize system reliability with limited energy budget. Through analysis, we found that lower levels of modular redundancy perform better in most cases on both energy savings and system reliability. The reasons partially come from the fact that, for a given number of processing units, lower levels of modular redundancy have more available modular redundant groups, which could explore an application’s parallelism better and result in more slack considering fixed application’s deadline. Thus, more energy savings could be obtained and more faults could be tolerated. Notice that, the optimal redundant configuration can be easily extended to event-driven rate based system as described in [97].

Finally, we address the trade-off between energy consumption and system reliability while considering the effects of energy management on fault rates. Based on previous published data, an exponential fault rate model for voltage scaling is proposed. A general framework to explore the trade-off between energy consumption and system reliability is considered. We illustrate that ignoring the effects of voltage scaling on fault rates may be too optimistic and will lead to unsatisfied system reliability when voltage scaling is used for energy savings.

In summary, the contributions of my doctoral work to the state of the art in energy management are as follows:

- Considering all the power consuming components in a system, a simple system power model is proposed and its effects on energy management are addressed [98]. Specifically, a minimum *energy efficient frequency* is developed when the system power has a constant component that can be efficiently removed by putting the system into a sleep state when it is idle. Using the power model, some theoretic bounds on energy savings of power management schemes are developed;
- Energy management schemes, both static and dynamic, are proposed for frame based real-time applications that run on shared-memory parallel systems [93, 94, 95, 100];
- A slack reservation scheme is proposed to efficiently incorporate the overhead of frequency changes into our energy management schemes. Moreover, discrete frequency/voltage levels and shared memory access contention are also addressed [95, 100].

- The concept of a *level of pessimism* for the number of expected faults is proposed and the optimal number of checkpoints for energy minimization or reliability maximization are considered. An energy efficient optimistic modular redundancy scheme (ONMR) is considered and the optimal frequency settings for all processing units are analyzed. Considering the parallel nature of the slack in parallel systems, an efficient adaptive parallel recovery scheme is proposed and used to determine optimal redundant configurations;
- Finally, we incorporate the effects of frequency/voltage scaling on the fault rates when studying the trade-off between reliability and energy management. Specifically, an exponential fault rate model for voltage scaling is proposed based on previous published data [96] and is used to study the trade-off between energy consumption and system reliability. To the best of our knowledge, this is the first work to consider the effects of energy management on reliability.

7.0 FUTURE WORK

Energy efficient fault tolerance for parallel systems is a relatively new research area and the problems we considered in this dissertation can be expanded in various directions. In what follows, we elaborate on promising extensions to our work.

In this dissertation, we have focused on frame-based real-time applications where tasks in an application share a common deadline and explored the fixed-priority list scheduling as the underlying scheduling algorithm for energy management. For more complicated applications, where tasks arrive at different times and have individual deadlines, instead of using the earliest ready longest task first (ER-LTF) heuristic for priority assignment, different heuristics (e.g., earliest deadline first (EDF)) may be needed. When applying slack reclamation for energy savings, the arrival time of tasks needs to be incorporated.

For better system utilization and for reducing the scheduling gaps caused by synchronization between tasks, preemption scheduling is needed, especially for periodic tasks. Although longer canonical schedules enable better slack sharing between processing units in parallel systems, the cost of creating a canonical schedule with the length of LCM (least common multiple) of tasks' periods is generally prohibitive. Instead, relatively longer dynamic schedules are preferred. Pfair is a well-known optimal scheduling algorithm for periodic tasks running on parallel systems [8]. However the dynamic schedule it generates is only one unit long since the algorithm needs to make scheduling decision for every time unit. We have proposed one boundary-fairness (Bfair) scheduling algorithm that is also optimal for periodic tasks on parallel systems and can generate relative longer dynamic schedules [99]. Combining Bfair with shared slack reclamation, we will explore energy management for periodic tasks on parallel systems.

We have ignored in this dissertation inter-processor communication because we focused

on shared-memory parallel systems. For distributed systems, where communication delays may be significant, task migration within different processing units could be prohibitive. Considering energy consumed by networking subsystems (e.g., wireless/sensor networks), which may also exhibit the trade-off between delay and energy consumption, it becomes more interesting but more complicated to explore slack time for energy savings. As shown in [60], without considering energy consumption for communication, our static power management with parallelism (SPM-P) can apply directly to distributed systems. Considering the trade-off between the energy consumption and latency in communication (e.g., wireless networks), we will further explore energy management for distributed systems.

When studying energy efficient fault tolerance using rollback recovery, we have focused on using the static slack for temporal redundancy. However, as we mentioned earlier, excessive amount of dynamic slack may exist in a system due to the run-time behavior of real-time applications. To explore dynamic slack for fault tolerance, we will extend the concept of reliability to that of an *expected reliability* that considers statistical run-time information about applications.

When considering the interplay of energy and reliability, the thermal effects of energy management on fault rates should be factored in addition to the effects of supply voltage on fault rates. Moreover, according to different working environments, *adaptive* energy efficient fault tolerance schemes can be explored. More resources (e.g., processing units or slack time) should be used for fault tolerance to achieve fixed reliability goals when fault rate increases due to, for example, busy cosmic ray for out-space applications.

BIBLIOGRAPHY

- [1] N. AbouGhazaleh, D. Mossé, B. R. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Proc. of Workshop on Compilers and Operating Systems for Low Power*, 2001.
- [2] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, Jun. 2000.
- [3] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proc. of the 7th International Workshop on Real-Time Computing Systems and Applications*, Dec. 2000.
- [4] J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, Jun 2001.
- [5] A. Avizienis and J. P. Kelly. Fault-tolerance by design diversity. *IEEE Computer*, 17:67–80, 1984.
- [6] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proc. of The 22th IEEE Real-Time Systems Symposium*, Dec. 2001.
- [7] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Optimal reward-based scheduling for periodic real-time systems. *IEEE Trans. on Computers*, 50(2):111–130, 2001.
- [8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.
- [9] S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proc. of The International Parallel Processing Symposium*, Apr. 1995.
- [10] A. Bertossi and L. Mancini. Scheduling algorithms for fault-tolerance in hard real-time systems. *Real Time Systems Journal*, 7, 1994.
- [11] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. *The case for power management in web servers*, chapter 1. Power Aware Computing. Plenum/Kluwer Publishers, 2002.

- [12] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger. Comparison of error rates in combinational and sequential logic. *IEEE Trans. on Nuclear Science*, 44(6):2209–2216, 1997.
- [13] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, Jan. 1995.
- [14] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, 35(11):1571–1580, 2000.
- [15] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Department of Computer Science, University of Wisconsin-Madison, Jun. 1997.
- [16] X. Castillo, S. McConnel, and D. Siewiorek. Derivation and calibration of a transient error reliability model. *IEEE Trans. on computers*, 31(7):658–671, 1982.
- [17] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: An approach for energy efficient computing. In *Proc. Int’l Symposium on Low-Power Electronic Devices*, 1996.
- [18] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuit*, 27(4):473–484, 1992.
- [19] Intel Corp. Mobile pentium iii processor-m datasheet. Order Number: 298340-002, Oct 2001.
- [20] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. On Software Engineering*, 15(12):1497–1505, 1989.
- [21] E. (Mootaz) Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proc. of Power Aware Computing Systems*, 2002.
- [22] E. (Mootaz) Elnozahy, R. Melhem, and D. Mossé. Energy-efficient duplex and tmr real-time systems. In *Proc. of The 23rd IEEE Real-Time Systems Symposium*, Dec. 2002.
- [23] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. of The International Conference on Computer-Aided Design*, pages 598–604, Nov. 1997.
- [24] X. Fan, C. Ellis, and A. Lebeck. The synergy between power-aware memory systems and processor voltage. In *Proc. of the Workshop on Power-Aware Computing Systems*, 2003.
- [25] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(3):272–284, 1997.

- [26] D. W. Gillies and J. W.-S. Liu. Scheduling tasks with and/or precedence constraints. *SIAM J. Compu.*, 24(4):797–810, 1995.
- [27] K. L. Gong and L. A. Rowe. Parallel mpeg-1 video encoding. In *Proc. of 1994 Picture Coding Symposium*, Sacramento, CA, Sep. 1994.
- [28] F. Gruian. System-level design methods for low-energy architectures containing variable voltage processors. In *Proc. of The Workshop on Power-Aware Computing Systems*, Nov. 2000.
- [29] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proc. of the 2001 International Symposium on Low Power Electronics and Design*, Aug. 2001.
- [30] F. Gruian and K. Kuchcinski. Low-energy directed architecture selection and task scheduling. In *Proc. of 25th IEEE Euromicro Conference*, pages 296–302, Sep 1999.
- [31] F. Gruian and K. Kuchcinski. Lenex: Task scheduling for low-energy systems using variable supply voltage processors. In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2001.
- [32] P. Hazucha and C. Svensson. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *IEEE Trans. on Nuclear Science*, 47(6):2586–2594, 2000.
- [33] P. Holman and J. Anderson. Guaranteeing pfair supertasks by reweighting. In *Proc. of the 22nd IEEE Real-Time Systems Symposium*, Dec. 2001.
- [34] <http://developer.intel.com/design/intelxscale/benchmarks.htm>, 2002.
- [35] <http://www.curvefit.com/>, 2004.
- [36] <http://www.darpa.mil/ipto/programs/pacc/>, 2004.
- [37] <http://www.transmeta.com>, 2001.
- [38] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proc. of The 14th Symposium on Discrete Algorithms*, 2003.
- [39] T. Ishihara and H. Yauura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of The 1998 International Symposium on Low Power Electronics and Design*, Aug. 1998.
- [40] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison Wesley, 1989.
- [41] T. Juhnke and H. Klar. Calculation of the soft error rate of submicron cmos logic circuits. *IEEE Journal of Solid-State Circuits*, 30(7):830–834, 1995.

- [42] K. M. Kavi, H. Y. Youn, and B. Shirazi. A performability model for soft real-time systems. In *Proc. of the Hawaii International Conference on System Sciences (HICSS)*, Jan. 1994.
- [43] A. Khemka and R. K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *Journal of Parallel and Distributed Computing*, 43:37–45, 1997.
- [44] D. Kirovski and M. Potkonjak. System-level synthesis of low-power hard real-time systems. In *Proc. of The Design Automation Conference*, Jun. 1997.
- [45] R. Koo and S. Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13(1):23–31, 1987.
- [46] C. M. Krishma and A. D. Singh. Reliability of checkpointed real-time systems using time redundancy. *IEEE Trans. on Reliability*, 42(3):427–435, 1993.
- [47] P. Kumar and M. Srivastava. Predictive strategies for low-power rtos scheduling. In *Proc. of the 2000 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Sep. 2000.
- [48] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.
- [49] H. Lee, H. Shin, and S. Min. Worst case timing requirement of real-time tasks with time redundancy. In *Proc. of Real-Time Computing Systems and Applications*, 1999.
- [50] F. Liberato, S. Lauzac, R. Melhem, and D. Mossé. Fault-tolerant real-time global scheduling on multiprocessors. In *Proc. of The 10th IEEE Euromicro Workshop in Real-Time Systems*, Jun. 1999.
- [51] F. Liberato, R. Melhem, and D. Mossé. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Trans. on Computers*, 49(9):906–914, 2000.
- [52] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients in combinational networks. In *Proc. of the 24th International Symposium on Fault-Tolerant Computing*, 1994.
- [53] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1):46–61, 1973.
- [54] J. Luo and N. K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proc. of International Conference on Computer Aided Design*, Nov. 2000.

- [55] J. Luo and N. K. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proc. of 15th International Conference on VLSI Design*, Jan. 2002.
- [56] A. P. Chandrakasan M. Srivastava and R. W. Brodersen. Predictive system shut-down and other architecture techniques for energy efficient programmable computation. *IEEE Trans. on VLSI Systems*, 4(1):42–55, 1996.
- [57] G. Manimaran and C. Siva Ram Murthy. A fault-tolerance dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Trans. on Parallel and Distributed Systems*, 9(11):1137–1152, 1998.
- [58] R. Melhem, N. AbouGhazaleh, H. Aydin, and Daniel Mossé. *Power Management Points in Power-Aware Real-Time Systems*, chapter 7, pages 127–152. Power Aware Computing. Plenum/Kluwer Publishers, 2002.
- [59] R. Melhem, D. Mossé, and E. (Mootaz) Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. on Computers*, 53(2):217–231, 2004.
- [60] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem. Energy aware scheduling for distributed real-time systems. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, Apr. 2003.
- [61] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating tasks on multiple resources. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, Dec. 1999.
- [62] D. Mossé, H. Aydin, B. R. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Proc. of Workshop on Compiler and OS for Low Power*, Oct. 2000.
- [63] Y. Oh and S. H. Son. Scheduling hard real-time tasks with 1-processor-fault-tolerance. Technical Report CS-93-27, Computer Science Department, University of Virginia, Charlottesville, VA 22903 USA, 1993.
- [64] A. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.
- [65] D-T. Peng. Performance bounds in list scheduling of redundant tasks on multiprocessors. In *Proc. of The 22th Annual International Symposium on Fault-Tolerant Computing*, Jul. 1992.
- [66] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. of Int’l Symposium on Low Power Electronics and Design*, Aug. 1998.

- [67] T. Pering, T. Burd, and R. Brodersen. Voltage scaling in the lpram microprocessor system. In *Proc. of Int'l Symposium on Low Power Electronics and Design*, 2000.
- [68] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct. 2001.
- [69] D. K. Pradhan. *Fault Tolerance Computing: Theory and Techniques*. Prentice Hall, 1986.
- [70] D. K. Pradhan and N. H. Vaidya. Roll-forward checkpointing scheme: A novel fault-tolerant architecture. *IEEE Trans. on Computers*, 43(10):1163–1174, 1994.
- [71] F. Quaglia and A. Santoro. Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Trans. on Parallel and Distributed Systems*, 14(6):593–610, 2003.
- [72] V. Raghunathan, P. Spanos, and M. B. Srivastava. Adaptive power-fidelity in energy aware wireless embedded systems. In *Proc. of The 21th IEEE Real-Time Systems Symposium*, Orlando, FL, Nov. 2000.
- [73] Rambus. RDRAM. <http://www.rambus.com/>, 1999.
- [74] B. Randell. System structure for software fault tolerance. *IEEE Trans. on Software Engineering*, 1(2):220–232, 1975.
- [75] J. A. Ratches, C. P. Walters, R. G. Buser, and B. D. Guenther. Aided and automatic target recognition based upon sensory inputs from image forming systems. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 19(9):1004–1019, 1997.
- [76] S. Saewong and R. Rajkumar. Practical voltage scaling for fixed-priority rt-systems. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.
- [77] N. Seifert, D. Moyer, N. Leland, and R. Hokinson. Historical trend in alpha-particle induced soft error rates of the alphaTM microprocessor. In *Proc. of the 39th Annual International Reliability Physics Symposium*, 2001.
- [78] Tezzaron Semiconductor. Soft errors in electronic memory: A white paper. available at <http://www.tachyonsemi.com/about/papers/>, 2004.
- [79] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. In *Proc. of the 24th IEEE Real-Time System Symposium*, 2003.
- [80] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design & Test of Computers*, 18(2):20–30, 2001.

- [81] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of the International Conference on Dependable Systems and Networks*, 2002.
- [82] A. Sinha and A. P. Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *Proc. of Design Automation Conference*, Jun 2001.
- [83] S. Thompson, P. Packan, and M. Bohr. Mos scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, Q3, 1998.
- [84] O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of The International Symposium on Low Power Electronics Design (ISLPED)*, Aug. 2002.
- [85] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proc. of The First USENIX Symposium on Operating Systems Design and Implementation*, Nov. 1994.
- [86] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Kerkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Design & Test of Computers*, 18(5):46–58, 2001.
- [87] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19:1321–1344, 1993.
- [88] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proc. of The 36th Annual Symposium on Foundations of Computer Science*, Oct. 1995.
- [89] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2003.
- [90] Y. Zhang and K. Chakrabarty. Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2004.
- [91] Y. Zhang, K. Chakrabarty, and V. Swaminathan. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Proc. of International Conference on Computer Aided Design*, Nov. 2003.
- [92] Y. Zhang, X. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proc. of The 39th Design Automation Conference*, Jun. 2002.
- [93] D. Zhu, N. AbouGhazaleh, D. Mossé, and R. Melhem. Power aware scheduling for and/or graphs in multi-processor real-time systems. In *Proc. of The Int'l Conference on Parallel Processing (ICPP)*, pages 593–601, Aug. 2002.

- [94] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Proc. of The 22th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2001.
- [95] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):686–700, 2003.
- [96] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. of the International Conference on Computer Aided Design (ICCAD)*, Nov. 2004.
- [97] D. Zhu, R. Melhem, and D. Mossé. Energy efficient redundant configuration for qos in reliable parallel servers. In *submitted to EDCC 2004*, 2004.
- [98] D. Zhu, R. Melhem, D. Mossé, and E.(Mootaz) Elnozahy. Analysis of an energy efficient optimistic tmr scheme. In *Proc. of the 10th International Conference on Parallel and Distributed Systems (ICPADS)*, Jul. 2004.
- [99] D. Zhu, D. Mossé, and R. Melhem. Periodic multiple resource scheduling problem: how much fairness is necessary. In *Proc. of The 24th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2003.
- [100] D. Zhu, D. Mossé, and R. Melhem. Power aware scheduling for and/or graphs in real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 15(9):849–864, 2004.
- [101] J. F. Ziegler. Terrestrial cosmic ray intensities. *IBM Journal of Research and Development*, 42(1):117–139, 1998.
- [102] J. F. Ziegler. Trends in electronic reliability: Effects of terrestrial cosmic rays. available at <http://www.srim.org/SER/SERTrends.htm>, 2004.